OBJECT-ORIENTED REAL-TIME

SIMULATION FOR A MANUFACTURING FACILITY,

A Thesis Presented to

The Faculty of The Russ College of Engineering and Technology

Ohio University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Somanath Lanka

November, 1994
ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Robert Judd, for his advice and support during my work. Working with him has been a learning experience and I have derived a lot of inspiration from him. I could not have completed this thesis without his help.

I also wish to thank members of the graduate committee, Dr. Nurre, Dr. Vassiliadis and Dr. Rabelo for the review of the thesis and giving suggestions for its improvement.

I also wish to thank my friends, Senthil, Sathya, Faiz, Shiva and others for giving the support and encouragement needed to see me through my graduate studies.

Finally, I offer my greatest thanks to my parents, without whose support and encouragement I would never made this far.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>ACKNOWLEDGEMENTS</th>
<th>LIST OF FIGURES</th>
<th>CHAPTER I</th>
<th>INTRODUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1.1 LITERATURE REVIEW</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.2 THESIS GOALS</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.3 THESIS OUTLINE</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER II</th>
<th>INTERPROCESS COMMUNICATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 ISO LAYERS</td>
<td>10</td>
</tr>
<tr>
<td>2.2 CONNECTION ORIENTED SERVICE</td>
<td>12</td>
</tr>
<tr>
<td>2.3 SOCKETS</td>
<td>13</td>
</tr>
<tr>
<td>2.5 CLIENT / SERVER MODEL</td>
<td>14</td>
</tr>
<tr>
<td>2.6 MESSAGE PASSING IN CIMLAB</td>
<td>15</td>
</tr>
<tr>
<td>2.6.1 COMMAND Message</td>
<td>18</td>
</tr>
<tr>
<td>2.6.2 REQUEST Message</td>
<td>19</td>
</tr>
<tr>
<td>2.6.3 REPLY Message</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER III</th>
<th>DESCRIPTION OF SIMAN INTERFACING</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 INTERFACE REQUIREMENTS</td>
<td>22</td>
</tr>
<tr>
<td>3.1.1 Sending Messages</td>
<td>22</td>
</tr>
<tr>
<td>3.1.2 Receive Messages</td>
<td>23</td>
</tr>
<tr>
<td>3.1.3 Entity Flow</td>
<td>23</td>
</tr>
<tr>
<td>3.2 USER INTERFACE</td>
<td>23</td>
</tr>
<tr>
<td>3.2.1 Sending COMMAND Message</td>
<td>24</td>
</tr>
<tr>
<td>3.2.2 Sending REQUEST message</td>
<td>26</td>
</tr>
<tr>
<td>3.2.3 Sending REPLY message</td>
<td>28</td>
</tr>
<tr>
<td>3.2.4 Receiving messages</td>
<td>29</td>
</tr>
<tr>
<td>3.2.4.1 Receiving COMMAND message</td>
<td>30</td>
</tr>
<tr>
<td>3.2.4.2 Receiving REQUEST message</td>
<td>30</td>
</tr>
<tr>
<td>3.2.4.3 Receiving REPLY message</td>
<td>31</td>
</tr>
</tbody>
</table>

| 3.3 INCOMING MESSAGES | 32 |
| 3.4 CONFIGURATION FILE | 32 |
| 3.5 EXPERIMENT FILE | 33 |
| 3.6 EXAMPLE | 33 |
4.1 IMPORTANT STRUCTURES .................................................. 35
4.2 COMMUNICATION ROUTINES .............................................. 39
  4.2.1 Routine writemsg .................................................. 40
  4.2.2 Routine readmsg .................................................. 41
  4.2.3 Routine establish .................................................. 42
4.3 USER ROUTINES ............................................................ 44
  4.3.1 Routine scan_buf .................................................. 44
  4.3.2 Routine getnoofids ............................................... 45
  4.3.3 Routine put_param ............................................... 47
  4.3.4 Routine variable_val ............................................ 48
4.4 SIMAN ROUTINES .......................................................... 48
  4.4.1 PRIME routine .................................................... 49
  4.4.2 WRAPUP routine .................................................. 51
  4.4.3 EVENT routine .................................................... 52
4.5 MESSAGE ROUTINES ...................................................... 54
  4.5.1 Routine send_cmd ................................................ 55
  4.5.2 Routine send_req ............................................... 55
  4.5.3 Routine send_rep ............................................... 57
  4.5.4 Routine process_msg ........................................... 58
  4.5.5 Routine receive_cmd ............................................ 58
  4.5.6 Routine receive_req ............................................ 59
  4.5.7 Routine receive_rep ............................................ 60
  4.5.8 Routine synchronize_time ..................................... 61
4.6 COMPILING .................................................................. 62

CHAPTER V
THE CIMLAB ........................................................................ 65
  5.1 CIMLAB DESCRIPTION .................................................. 65
    5.1.1 CNC Machines ..................................................... 65
    5.1.2 Gm Fanuc's S-10 Robot ......................................... 65
    5.1.3 Feeders & Bin ..................................................... 68
    5.1.4 Computer ........................................................... 68
  5.2 IMPLEMENTATION .......................................................... 69

CHAPTER VI
DESCRIPTION OF SIMULATION MODEL .................................... 70
  6.1 Description of the CIMLAB ............................................. 70
  6.2 ROBOT ..................................................................... 71
    6.2.1 ROBOT Behavior .................................................. 71
      6.2.1.1 GOTO message .............................................. 72
      6.2.1.2 PLACE message .......................................... 72
      6.2.1.3 PICK message .............................................. 73
  6.1.2 SIMAN Model .......................................................... 73
    6.2.2.1 SIMAN Transporter .......................................... 74
| C.1.2 | millmodule.mod | 102 |
| C.1.3 | lethemodule.mod | 103 |
| C.1.4 | robotmodule.mod | 104 |
| C.1.5 | cimlab.exp | 110 |
| C.2.1 | inet.h | 115 |
| C.2.2 | register.h | 117 |
| C.2.3 | cimlab.h | 123 |
| C.2.4 | cimlab.lib | 125 |
| C.2.5 | cimlab.c | 129 |
| C.2.6 | msg.lib | 134 |
| C.2.7 | makefile | 140 |
LIST OF FIGURES

Figure 1.1  Simulation in Testing Phase .................................. 4
Figure 1.2  Simulation in Implementation Phase ...................... 5
Figure 2.1  Functional Layers of OSI model ............................ 11
Figure 2.2  Message Passing From CIMLAB ............................ 16
Figure 5.1  CIMLAB Layout .................................................. 66
Figure 5.2  Mill Machine ..................................................... 67
Figure 5.3  Lathe Machine ................................................... 67
Figure 5.4  Gm-Fanuc S-10 Robot ......................................... 68
Figure 6.1  Controller Sending and Receiving Messages ............. 71
Figure 6.2  Positions in CIMLAB .......................................... 72
Figure 6.3  Messages in ROBOT Model .................................. 73
Figure 6.4  Messages in Machine Model ................................. 81
CHAPTER I

INTRODUCTION

Simulation has been well recognized as a tool for the design and evaluation of complex advanced manufacturing facilities, such as highly automated production lines and Flexible Manufacturing Systems (FMS). Simulation is used to predict the performance of the system between alternative machine layouts and scheduling rules; therefore, to determine the best performance system. This system can be tested with the anticipated production schedules to determine if the planned production capacity is achieved with good utilization of resources. Any limitations in the system can be identified, and the model can be adapted to correct problems. Thus, major modifications to a large capital investment can be avoided after installation.

Simulation can also have a significant role in the analysis of existing manufacturing facilities. Simulation can analyze the performance of existing facilities to identify problems, such as high inventory and poor resource utilization. The simulation is equally effective at analyzing manual operations, such as highly automated operations. It is often useful to study a dynamic real-world system to learn something about its behavior. It is necessary to build a model to study the performance of the system, since experimentation with the system itself is disruptive, not cost-effective, or simply impossible (e.g., the manufacturing facility that has not yet been built). For example, a manufacturing firm that is contemplating building a large extension onto one of its plants
may be unsure about whether the potential gain in the productivity cost would justify the construction cost. An appropriate model could shed some light on this question by comparing the operation of the plant as it currently exists and as it would be if the plant is expanded.

If the relationships that compose the model are simple enough (e.g., one machine), it may be possible to use mathematical methods to answer the question "what-if." Unfortunately, most of the manufacturing systems, like the Computer Integrated Manufacturing Laboratory (CIMLAB) are too complex to allow realistic models to be evaluated mathematically. These models must be studied by means of simulation. In a simulation we use computers to evaluate a model numerically over a time of interest, and data is gathered to estimate the true characteristics of the model. Using this data, perturbation analysis can be done to estimate the characteristics of the model when expanded.

Discrete event simulation has been characterized by two significant trends in recent years. The first trend is the increasing use of the simulation with complex manufacturing systems. The second has been the increasing use of computer graphics for animated displays of the simulated system.

Many recent applications of automation have been using a technology known as the Flexible Manufacturing System. A FMS generally employs Computer Numeric Controller (CNC) machine tools with automated material handling of parts and tooling. The entire system is computer controlled.

A FMS implementation requires substantial study to optimize design and
performance. Traditional mathematical techniques do not provide the opportunity to gain experience with the hardware aspects of the system.

Simulation as a tool permits management to define objectives and examine how well a proposed design will meet the objective through the total system balance. The decision-maker must be careful to purchase a solution that is compatible with the existing disciplines and equipment. The Computer Integrated Manufacturing (CIM) philosophy must function with a balanced methodology to eliminate the possibilities in manufacturing industries. Computer simulation will permit the evaluation of new automation alternatives. Therefore, simulation plays an important role in CIM implementation.

It has been suggested that a manufacturing system pass through four phases during its life cycles. Simulation has a role to play in each of these phases:

1) **Design** Typical issues for which simulation can be used during the design phase are throughput, resource, requirements and control logic inventory levels.

2) **Justification** Typical phase issues here are: To Prove the existing system is inadequate and that the proposed system will do the job.

3) **Implementation** Implementation phase issues are design refinements and integration into existing operations.

4) **Operation** Typical issues here are determination of corrective actions in response to the unforeseen events.

The simulation is used as a testing field, for education, and for visualization as a control station component.
were not realized in the design phase. A longer period for the implementation is now needed. Here, the simulation can be used to help test the new control algorithms. Simulation also can be employed when the real system is functional (Figure 1.2). In this situation, simulation is interfaced with the control software and the real system. The advantage is that the simulation takes place at real time and the simulation results can be used to monitor the system; for example, when an existing system has breakdowns or is changed, the simulation can be used to study the behavior of the system under those conditions.

1.1 LITERATURE REVIEW

Felix Chan and Adrian Smith (1993) discussed the techniques to evaluate the performance of an assembly line that requires modifications to achieve integrated manufacturing. The simulation of the existing assembly line confirmed the operational problems previously identified and allowed validation of the computer model. The
computer model was then altered to evaluate the performance of several alternative modified assembly lines that would avoid the present operational problems. They have discussed the benefits of simulation in analyzing the manufacturing systems.

Thompson (1991) discussed the importance of simulation software as a tool to design manufacturing systems. He described the importance of the simulation throughout the life of the system: from building it, to implementing it, and even in operating the system daily. He discussed the importance of simulation in every phase of the automation, namely, planning, design, implementation and operation.

Hitchens (1984) discussed the advantages of the real-time simulators. He described the real-time simulator as capable of reacting to stimuli at the same rate as the real system. The simulator allows people and computers to interact with the simulated system. A real-time simulator allows playing "what-if" games with new algorithms to observe results. The simulator can be connected to computers and programmable controllers to test software, and can replace robots and conveyors to test the overall control system.

Cheng (1985), in his paper discussed the importance of simulation in a Flexible Manufacturing System. In using simulation to evaluate the Flexible Manufacturing Systems’ performance, he used two levels of system definitions: namely, macro and micro for gross and detailed simulation results, respectively. He described the importance of microcomputer-based simulator aimed at providing industrial designers with a rapid and cost-effective aid for FMS design and appraisal.

A simulation model capable of representing a large variety of Flexible
Manufacturing Systems has been developed by Manalis, Bilalis, and Konstantinidis (1987). The model can be used for design and control stages of the FMS development. They described the different modules of the simulation model. The model's validation is examined by comparing it with a model written in a specialized simulation language.

In CIM environment, the control facility plays an important role. Bilberg and Alting (1991) described a new approach in the development of the control software. The concept is based on a simulation package serving two functions: the simulation of facility and aid in the control design. He described the common element of the simulation and control as decision maker, statistical calculator and animation generator. First, a rough simulation for system analyses should be developed, and then a detailed simulation model should be developed for the optimizing control logic. The simulation model should be converted to a control program and tested using emulators, and then it can be implemented in FMS.

The CIM strategy has been suggested as the business philosophy that will eliminate the piecemeal approaches attempted in the past. Good communications across the whole company are needed to develop alternative designs that can be accepted and evaluated through the creation of a common data base. This can be achieved through functional modeling, information modeling and computer simulation models. A disciplined approach to planning can best represent the desired goals of that particular business. An effective planning methodology will, therefore, be the vehicle for the company to safely approach a CIM strategy without disrupting the ongoing production operations.
1.2 THESIS GOALS

In recent years, there are systems that are pre-processors geared to producing simulation models for specific problem classes. In a sense, this makes the system vendor, the modeler, and the buyer the model users. With these simulation systems, a particular class of problems can be modeled with relative ease, but the modeler is constrained to stay within that class. At the opposite end of the spectrum is the general purpose simulation system in which almost any problem can be modeled, but more expertise and effort are required. The tradeoff is between modeling ease and modeling flexibility. Since CIMLAB is a complex manufacturing system and in the future the number of machines in the CIMLAB may increase, the software to be used should have modeling flexibility and user interface capability and should function in real-time. The simulation software used in CIMLAB is SIMAN. CINEMA is an animation system and makes the animation displayable concurrent with the simulation model developed in SIMAN. Hence, CINEMA is used to animate the CIMLAB along with SIMAN.

In order to achieve the modeling flexibility the proposed simulation model must be integrated with the control software and with the real machines. The proposed simulation model should verify the control algorithms. The simulation model is to be based on modules. A module in the simulation model is a set of actions performed by the generic CNC machines or a generic robot. These modules in the simulation model must be independent of each other. Every module in the simulation model should communicate with the control software. In the proposed simulation model the user should not worry about the integration of the softwares. Using the modules, the user can model
any type of manufacturing system following certain rules. This simulation is to provide a real-time animation of the actual status of the system.

1.3 THESIS OUTLINE

Chapter 2 describes about the interprocess communication between the computers. The message format employed in CIMLAB is also discussed. A detailed description about modeling of SIMAN programs to interface with the control software tool is described in Chapter 3. The detailed description of the SIMAN C interfacing program with the control software FLEXIS is described in Chapter 4. The description of CIMLAB is described in Chapter 5. The design and implementation of the CIMLAB simulation model is described in Chapter 6. The conclusions and further recommendations are discussed in Chapter 7.
CHAPTER II
INTERPROCESS COMMUNICATION

For two computers to communicate with each other, protocols, message formats are needed. In CIMLAB the simulation software, SIMAN, and the control software tool, FLEXIS, are interfaced. These softwares run on different machines. A server is used to facilitate communication between them. A particular message format was developed for the two softwares. This chapter describes the International Standards Organizations (ISO) layers, the Transmission Control Protocol (TCP/IP), sockets and client server architecture [12]. The message format used in the CIMLAB is described later in this chapter.

2.1 ISO LAYERS

The International Standards Organization (ISO) has developed an Open System Interconnection (OSI) model that divides the task of computer communications into seven functional layers. The first four OSI layers form the lower-layer infrastructure of the OSI model [12] (Figure 2.1). These layers provide the end to end services responsible for data transfer. The remaining form the upper layer infrastructure of the OSI model. These layers provide the application services for information transfer.

**Physical Layer**
This layer is responsible for the electro-mechanical interface to the communication's media.

**Data-Link Layer**
This layer provides the exchange of data between the network layer entities. It detects and corrects any errors that may occur in the
physical layer transmission. It is responsible for transmission, framing and error control over a single communications link.

**Figure 2.1 The Functional Layers of OSI Reference Model**

**Network Layer**

The network layer manages the operation of the network. In particular, it is responsible for the routing and management of data exchange between transport layers within the network. This layer is responsible for data transfer across the network, independent of both the media comprising the underlying subnetworks and the topology of those subnetworks.

**Transport Layer**

This layer is responsible for reliability and multiplexing of data transfer across the network (in addition to that provided by the
network layer) to the level required by the application.

Session Layer  The Session Layer provides the services needed by presentation layer entities that enable them to organize. It also synchronizes their dialogue and manages the data achieved. This layer is responsible for adding control mechanisms to the data exchange.

Presentation Layer  The Presentation Layer manages the representation of information that Application Layer entities either communicate or reference in their communication. This layer is responsible for adding structure to the units of data that are being exchanged.

Application Layer  This is responsible for managing the computer communications between applications.

2.2 CONNECTION ORIENTED SERVICE

In OSI, the services offered by a layer are either connection-oriented or connection-less. A connection-oriented mode has three distinct phases:

Connection Establishment  The users and the provider negotiate the way in which the service will be used. If successful, this results in a connection being established, which is an explicit binding between the two service users.

Data Transfer  The service users exchange data.

Connection Release  The binding between the service users is discarded.

OSI application services are connection-oriented. The underlying end-to-end layers (i.e., Transport, Network, Data link, Physical) are connection-oriented services, but may
be internally composed of connection-less-mode protocols. The transport layer is important, because it is the lowest layer in the reference model that provides the basic service of reliable, end-to-end data transfer needed by applications and higher layer protocols. The transport layer defines a set of services common to layers of many contemporary protocol suites, including Transmission Control Protocol (TCP/IP).

2.3 SOCKETS

The basic building block for communication between two services is the socket [12]. A socket is an endpoint of communication to which a name may be bound. Each socket in use has a type and one or more associated processes. Sockets are typed according to the communication properties visible to a user. Two processes can communicate only between sockets of the same type. There are several types of sockets currently available:

Stream Socket This supports the bi-directional, reliable, sequenced and unduplicated flow of data.

Datagram Socket This supports the bi-directional flow of data that is not promised to be sequenced, reliable or unduplicated.

When two processes communicate, the sockets must be connected. To initiate a connection, sockets should be created using the function socket(domain, type, protocol). A socket is created without a name. Until a name is bound to a socket, processes have no way to reference it, and, consequently no messages may be received on it. Communicating processes are bound by an association. The routine bind(s, name, namelen) system call can be used for binding a name to the socket.
Connection establishment is usually asymmetric, with one process a client and the other a server.

### 2.5 CLIENT / SERVER MODEL

The client/server model is the most commonly used in constructing distributed applications [12]. In this scheme, client applications request services from a server process. The client and server require a well-known set of conventions before service may be rendered. This set of conventions comprise a protocol that must be implemented at both ends of a connection. Depending on the situation, the protocol may be symmetric or asymmetric. In an asymmetric protocol, either side may play the master or slave roles. In a symmetric protocol, one side is immutably recognized as the master, with the other as the slave.

The server, when willing to offer services to the clients, binds a socket using routine `bind()` to a well-known address associated with the service, and then passively listens using the routine `listen()` set to the length of the connection queue, and then enters the loop. Inside the loop, the server calls routine `accept()` to wait until the next connection request arrives, uses routines `read()` and `write()` to interact with the client, and finally uses `close()` to terminate the connection. The server then returns to the `accept()` call, where it waits for the next connection.

The client requests services from the server by initiating a connection to the server's socket. On the client side, the `connect()` call is used to initiate a connection. If the client's process socket is unbound at the time of the connect call, the system will bind a name to the socket, if necessary.
The client then interacts with the server using `write()` system call to send the request and `read()` system call to receive the replies.

The advantage of client-server model is its adaptability to use in distributed systems. If a client communicates by sending it messages, the client need not know whether the message is handled locally in its own machine, or whether it was sent across a network to a server on a remote machine.

As far as the client is concerned, the same thing happens in both cases: a request is sent and a reply comes back. Therefore, messages can be used for communicating between two clients with neither client needing to know how the messages are handled by the server.

2.6 MESSAGE PASSING IN CIMLAB

As with the method of execution, highly concurrent communication in the real world is usually implemented by scheduling messages on predominantly sequential media. Interactions via messages involves two main actions: 1) binding the message to a particular format or method, and 2) transferring the message.

There are three phases to message transfer: 1) The software, a client, resolves the object and method name, packs parameters or message content in a standard format and introduces the message into transmission medium; 2) The message propagates over the medium; 3) The message is received by another client and unpacked. The above message passing system is employed in CIMLAB. The simulation software SIMAN, has three different actors or modules. Each module is in no way related to any other module. In the FLEXIS control tool set, all actors are independent from each other. The messages
that travel between these two software packages have a standard format (discussed later in this chapter). These two packages communicate with each other via a server using the standard messages. The communication between the software packages in CIMLAB takes place in two layers (Figure 2.2). The first layer is between device to device and the second layer is between device to actor.

![Diagram of CIMLAB message passing](image)

Figure 2.2 Message Passing from CIMLAB

When a message passes from one software to another, it first passes from source
actor to source device and from source-device to destination-device and then from to
destination actor. The format of a message that communicates between two actors in
device-to-device layer is

<table>
<thead>
<tr>
<th>Length</th>
<th>Destination</th>
<th>Source</th>
<th>Action</th>
<th>Actor message</th>
</tr>
</thead>
</table>

where **Length** is the total number of bytes in the message.

**Dest** is the ID of the destination device.

**Source** is the ID of the source device of the message.

**Action** is the type of the message. Message types are defined in the file
inet.h. **GO** is defined as 1. The server issues this type of message
when all its clients are registered.

**KILL** is defined as 9. When the client or server finds something
unusual in CIMLAB, they issue the KILL message, which is
propagated to all clients. When a client receives a KILL it
terminates itself.

**NORMAL** is defined as 0. Normal messages are the messages that
are sent by the actors to perform a specific action in CIMLAB.

**Actor Msg** is the actual contents of the message. It contains the type of
message, i.e., COMMAND, REQUEST, REPLY. The format of
each of these message types are given in the next section.

The three types of messages that are used for communication between control
Software Flexis and the Simulation Software SIMAN are
2.6.1 COMMAND Message

A COMMAND is a type of message given by an actor to another actor to do a specific action. The destination actor does not need to REPLY to this message. The information that is sent with the command message is the type of the message, message ID of the actor and a character string.

**Type**
- It is defined as the type of message that is coming from the source actor.

**msgid**
- The identifier for the message. The Msgid describes the type of action taken by the destination source after receiving the message.
  - It is an integer.

**Param**
- The parameters that are being sent in the command, formatted as a string. These parameters can be attributes, variables or file names.

When a SIMAN actor sends a COMMAND message to another actor, the actual message string is created using a C function:

```c
sprintf(contents,"%4d %4d %s",CMD, msgid, param);
```

The message is directed to the server using the C subroutine:

```c
writemsg(dest,Source,NORMAL,contents)
```

When the message reaches the server, it reads the destination and type of the message. The server finds the corresponding device for the destination actor and sends
the message to the destination actor. When the message COMMAND is sent, the source actor of the message will not wait for the reply from the destination actor of the message to finish the job. The source actor can send another COMMAND message to the same destination actor, even when the destination actor is processing the first COMMAND message.

2.6.2 REQUEST Message

A REQUEST is a type of message, such as a request by an actor to another actor to do a specific action and wait for the reply. This type of message sends a request to the specified destination. The information that is sent with the request message is destination ID, MSGID, Transaction number and param.

Type

REQUEST.

Msgid

The identifier for the message. The Msgid describes the type of action taken by the destination source after receiving the message. It is an integer.

Transaction no.

The Transaction no. is used for giving reply to the source actor of the message. It acts as an identifier for the source actor, when a reply comes from the destination actor. It is an integer.

param

The parameters that are being sent in the request, as a character string. These parameters can be attributes, variables or file names.

When a SIMAN actor sends a REQUEST message to another actor, the actual message string is created using a C function:
The message is directed to the server using the C subroutine:

\begin{verbatim}
void writemesg (DestId, SourceId, NORMAL, contents) {
    // code for writing the message to the server
}
\end{verbatim}

When the message reaches the server, it reads the destination and type of the message. The server finds the corresponding device for the destination actor and sends the message to the destination actor. When the request message is sent, the source actor of the message will wait for the reply from the destination actor of the message to finish the job. The source cannot send another REQUEST or COMMAND message to the same destination actor, when the destination actor is processing the first REQUEST message.

2.6.3 REPLY Message

This message is a reply to a previous REQUEST. The information that is sent with the request message is Destination ID, Msgid, Transaction no. and param. The destination actor sends the reply message for the request message. The reply message contains the transaction number so that the source actor can match it to a proper REQUEST.

- **Type**: COMMAND
- **Msgid**: The identifier for the message. The Msgid describes the type of action taken by the destination source after receiving the message. It is an integer.
- **Transaction no.**: The transaction number for a REPLY is equal to the transaction number of the REQUEST which generated the REPLY. This is used by destination actor to match the REPLY to the pending REQUEST.
The parameters that are being sent in the request, as a character string. These parameters can be attributes, variables or file names.

When a SIMAN actor sends a REQUEST message to another actor, the actual message string is created using a C function:

```c
sprintf(contents,"%4d %4d %4d %s",REP, msgid, tn, param);
```

The message is directed to the server using the C subroutine:

```c
writemsg(Dest,Source,NORMAL,contents)
```

When the message reaches the server, the server finds the corresponding device for the destination actor and sends the REPLY to the source actor. When the reply message is sent, the source actor of the reply message will not wait for any message from the destination actor of the message.
CHAPTER III
DESCRIPTION OF SIMAN INTERFACING

This chapter describes the interface of SIMAN simulation language to the control software. These are the things for a SIMAN modeler to know in order to communicate with FLEXIS, such as interface requirements and sending and receiving messages, which are discussed in detail in this chapter. The configuration file and experiment file format are also discussed. An example is provided at the end of this chapter to demonstrate how messages pass between the devices.

3.1 INTERFACE REQUIREMENTS

SIMAN simulation models have to communicate with the control software. To communicate with the FLEXIS toolset, the SIMAN simulation model must be able to send and receive the messages. The SIMAN simulation model should move the entities in the model to process the message or to send the message to the control software. It uses TCP/IP routines in the CEVENT routine to communicate with the FLEXIS toolset. This is discussed in Chapter 4.

3.1.1 Sending Messages

In order to send the messages, SIMAN simulation models need to specify the destination of the message and the type of action to be performed by the destination actor. The destination actor ID, MSGID, SOURCE, NUMPARAMS and PARAMS are assigned to the global variables before sending the message to the destination actor. The
simulation model sends the message when the entity enters the EVENT block in the model file.

3.1.2 Receive Messages

When a SIMAN client receives the message, it creates an entity. It then assigns the message content as attributes to the entity and sends the entity to the label in the simulation model. The SIMAN simulation model has to send a REPLY message for every REQUEST message received. The SIMAN model should process the message by passing the entity through the appropriate blocks and by using the attributes of the entity. The labels and the type of the message must be declared in the configuration file.

3.1.3 Entity Flow

The entity which enters the model when a message arrives can process the message content. This entity can also send a signal to allow another entity in the model. In the latter case, the entity may pass its attributes to another entity. The arriving entity can be disposed once it processes the message, or it can be used for other simulation purposes.

3.2 USER INTERFACE

The SIMAN model sends the message by routing an entity to an event block. The event number is used to distinguish the type of message to be sent from the simulation model. The COMMAND messages use event number 1. The REQUEST messages use event number 2. The REPLY messages use event number 3. A fourth event is used to process the incoming messages from the FLEXIS toolset. (A fourth event is discussed in
3.2.1 Sending COMMAND Message

In order to send a COMMAND message to the FLEXIS toolset, an entity must initialize the values of the global variables, MSGID, SOURCE, NUMPARAMS, PARAMS and DEST. The entity must then enter the EVENT block. The event number to be used for COMMAND message is 1.

**Msgid**
The type of action to be performed by the destination actor upon receiving the message. The value of this variable should be an integer and is predefined. The Msgid's used in the simulation model should be declared in system file. The following Msgid's are pre-defined for the CIMLAB application.

<table>
<thead>
<tr>
<th>Msgid</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUN</td>
<td>1</td>
<td>To run the machine</td>
</tr>
<tr>
<td>LOAD_PROG</td>
<td>2</td>
<td>To load the CNC program</td>
</tr>
<tr>
<td>GOTO</td>
<td>1</td>
<td>To move the robot</td>
</tr>
<tr>
<td>PICK</td>
<td>2</td>
<td>To load the part</td>
</tr>
<tr>
<td>PLACE</td>
<td>3</td>
<td>To unload the part</td>
</tr>
<tr>
<td>DIO</td>
<td>4</td>
<td>To open the door of the machines</td>
</tr>
</tbody>
</table>

**Source**
The variable Source contains the ID of the source actor of the message. The value of the variable is an integer and must be defined in the configuration file. The Source IDs of the actors in CIMLAB are

```plaintext
    MILL      201
    LATHE     202
    ROBOT     203
```

**Dest**
The destination actor ID of the message. The value of the destination actor ID is an integer and must be defined in the system file.
IDs used for the CIMLAB are:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MILL</td>
<td>1</td>
</tr>
<tr>
<td>LATHE</td>
<td>2</td>
</tr>
<tr>
<td>ROBOT</td>
<td>9</td>
</tr>
</tbody>
</table>

**Numparams**
The number of additional variables needed to send the message to the FLEXIS. The value of the Numparams is usually zero. If the parameters are to be sent, then approximate entity attributes need to be set.

The names of these global variables cannot be changed, and they must be properly declared in the experiment file. After sending the COMMAND message, the entity returns to the EVENT block in the model file and goes down the simulation model to process the model.

The following code sends the COMMAND message to the FLEXIS Robot actor. The SIMAN source actor ID is assumed to be 203, FLEXIS destination actor ID is assumed to be 9, type of action to be performed is GOTO (moving the ROBOT from one station to another station). The global variables, SOURCE, DEST, MSGID, NUMPARAMS are set to the above values. Since there are no additional parameters to be sent with the message, the value of the NUMPARAMS are set to zero. The values of the SIMAN source actor ID, FLEXIS destination actor ID and message ID are defined in the system file.

**Create:** Creates an entity in the model file

; Sending the COMMAND message to the FLEXIS

**Assign:**
Source=203:
Dest=9:
Msgid=1:
Numparams=0;
Event, 1;          Event 1 sends the COMMAND message.
;  continue the simulation program.

3.2.2 Sending REQUEST message

In order to send the REQUEST message from the SIMAN simulation model, an entity must set the global variables, MSGID, SOURCE, DEST, SIG, and NUMPARAMS. Then the entity must be sent to the EVENT block. The event number to be used is 2. After sending the REQUEST message, the entity must enter the QUEUE block. The hold block following the QUEUE block must be a WAIT block, and the entity must wait in the WAIT block.

**MSGID**

The type of action to be performed by the destination actor upon receiving the message. The value of this variable should be an integer and is predefined in system file.

**SOURCE**

The ID of the source actor of the message. The value of the variable is an integer and should be defined in the configuration file.

**DEST**

The destination actor ID of the message. The value of the destination actor ID is an integer and must be defined in the system file.

**NUMPARAMS**

The number of additional variables needed to be sent to destination actor. The value of the Numparams is usually zero. If the parameters are to be sent, then the approximate entity attributes need to be sent.
SIG

The signal number to be used for releasing the entity in the SIMAN model when the REPLY arrives. The value of the SIG variable is an integer and must be the signal code used in the WAIT block which follows the QUEUE block. The signal code should be different for different REQUEST messages.

The names of the global variables cannot be changed and should be declared in the experiment file. After sending the REQUEST message from the EVENT block, the entity returns and waits in the QUEUE block until the WAIT block is satisfied by a signal generated by the REPLY.

The following code sends the REQUEST message to the FLEXIS robot actor. The SIMAN source actor ID is assumed to be 203, the FLEXIS destination actor ID is 9, type of action to be performed is GOTO and the value of the signal code to be used by FLEXIS in REPLY is assumed to be 1. The global variables, SOURCE, DEST, MSGID, NUMPARAMS and SIG are set to the above values. Since there are no additional parameters to be sent with the message, the value of the NUMPARAMS is set to zero. The values of the source actor ID, destination actor ID and message ID are defined in the system file.

```
create;
Simulation program;
Assign: Source=203;  ! Source ID
Dest=9;    ! Destination ID
Msgid=1;   ! Message ID
SIG=1;     ! Signal code
Numparams=0; Number of parameters

Event,2; Event to send REQUEST messages
```
3.2.3 Sending REPLY message

When a REQUEST message arrives from the FLEXIS, a REPLY has to be sent back to the source actor. Normally, the entity created by receiving the REQUEST message can be used to send the REPLY back. The SIMAN model has to complete processing of the REQUEST message before it sends a REPLY message. To send a REPLY, the entity must assign the values to the global variables, SOURCE, DEST, MSGID, and NUMPARAMS. Then the entity must be sent to the EVENT block. The event number to be used is 3.

**MSGID**
The type of action to be performed by the destination actor upon receiving the message from the simulation model. The value of this variable should be an integer and is predefined.

**SOURCE**
The ID of the source actor of the message. The value of the variable is an integer and should be defined in the configuration file.

**DEST**
The destination actor ID of the message. The value of the destination actor ID is an integer and must be defined in the system file.

**NUMPARAMS**
The number of additions variables needed to send the message to the FLEXIS. The value of the NUMPARAMS is usually zero. If the additional parameters are to be sent, then the variables declared in the simulation model should be set.
The SIMAN interface program described in Chapter 4 sets the transaction number of the corresponding REQUEST message for the REPLY. The names of the global variables cannot be changed and must be declared in the experiment file. After sending the REPLY, the entity returns to the model and is usually disposed.

The following code sends the REPLY message to the previous REQUEST message sent by the destination actor (described in Section 3.2.4.2). The SIMAN source actor ID is assumed to be 203, the destination actor ID is assumed to be 9. The global variables, SOURCE, DEST, MSGID, and NUMPARAMS are set to the above values. Since there are no additional parameters to be sent with the message, the value of the NUMPARAMS are set to zero. The values of the source actor ID, destination actor ID and message ID are defined in the system file.

```
Create; label Station, Enter; REQUEST message enters model Simulation program;

; processed the simulation model ready to give REPLY
Assign: Source=203:
    Msgid=1:
    Numparams=0;

Event,3: Dispose; Enters REPLY event and sends REPLY
```

### 3.2.4 Receiving messages

The SIMAN simulation program can receive all three types of messages. To process the incoming messages efficiently, the simulation model should be developed in modules. A module in a simulation model represents a workcell or a department of a larger facility. Each module should have a set of labelled blocks which defines the entry
point for the incoming message. The label of the module, along with the type of action the module performs (MSGID) and ID of the module must be properly declared in the configuration file.

When a message arrives to the SIMAN model, an entity is sent to the appropriate labelled block in the module with the message parameters placed in its entity attributes. The incoming message is processed by sending the entity through the appropriate SIMAN blocks.

3.2.4.1 Receiving COMMAND message

When a COMMAND message arrives, an entity enters the simulation model file through a labelled block described in the configuration file. The entity processes the message, and then the entity should be disposed.

The following code processes the COMMAND message: The entity enters the model through the label ENTERMILL. The label is declared in the configuration file. The entity then processes the message using the SIMAN blocks. The attribute of the entity is the parameter which is used for delay time. The entity is disposed after processing the message.

ENTERMILL

<table>
<thead>
<tr>
<th>Queue</th>
<th>MachineQ; COMMAND messages arrives with an entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seize:</td>
<td>Machine; Seizes the machine</td>
</tr>
<tr>
<td>Delay:</td>
<td>A(1); Uses the attribute of the entity to delay</td>
</tr>
<tr>
<td>Release:</td>
<td>Machine; Releases the machine Dispose;</td>
</tr>
</tbody>
</table>

3.2.4.2 Receiving REQUEST message

When a REQUEST message arrives, the entity enters the model through a labelled
block. The entity processes the message, then sends the REPLY to the source of the REQUEST message, and finally is usually disposed. (sending a REPLY message is described in Section 3.2.3).

The following code processes the REQUEST message FLEXIS MILL actor: An entity enters into the model through a label RUNMILL. The label RUNMILL and message ID is declared in the configuration file. The attribute used in the model file is a parameter of the entity. After processing the message it returns the REPLY message and is disposed.

```
RUNMILL  Queue; MachineQ; REQUEST arrives with an entity
Seize:    Machine; Seizes the machine
Delay:    A(1); Uses the attribute for delay
Release   Machine; Releases the machine

; processed the simulation model sends the REPLY
Assign:   Source=203
         Msgid=1:
         Numparams=0;
Event,    3: !Enters REPLY event and sends REPLY
Dispose;
```

3.2.4.3 Receiving REPLY message

When a REQUEST message is issued, a REPLY message must arrive at some time. As described in Section 3.2.2, the entity which sent the REQUEST for the REPLY must wait in the WAIT block for a signal. When the REPLY arrives, an appropriate signal is sent to the WAIT block and the entity waiting in the QUEUE block is released and continues down the model. Since SIMAN is used to model the physical devices, it normally does not send requests or receive replies.
3.3 INCOMING MESSAGES

When simulation starts, the PRIME routine opens a socket and establishes connection with the server. An entity is created in the PRIME and initially schedules the entity to event 4. The SIMAN reads the messages sent to it by scheduling an entity to trigger event 4 (described in Chapter 6). When event 4 is triggered, SIMAN reads the socket and routes the incoming messages to appropriate labels in the model. It then reschedules the entity to trigger the event 4. The SIMAN clock is tied to the real-time. The SIMAN process is put to sleep in the event 4 until the cpu clock catches the simulation time. The SIMAN events 1, 2, 3 sends the messages to the FLEXIS toolset. The routine wrap closes the socket and exits from the simulation program, after printing the SIMAN summary report.

3.4 CONFIGURATION FILE

The configuration file contains actor IDs, message IDs and the labels of the model file. The labels are used to send the entities to the proper block in the model file, when the message arrives at the simulation model. The actor IDS are used to register with the server, while connecting to the server. When the simulation is started, the PRIME routine reads the configuration file and stores the contents of the configuration file in the appropriate variables. The format for the configuration file is

```plaintext
<maximum actor ID>, <minimum actorID>
<actor1 ID>  <actor2 ID>  <actor3 ID>  -1 -1 is to indicate the end of IDS
<actor1 ID>  <Msgid>  label
<actor2 ID>  <Msgid>  label
<actor3 ID>  <Msgid>  label
```

The following is the sample code of the configuration file of the CIMLAB application.
203 201
201 202 203 -1
201 1 MILLRUN
202 1 LATHERUN
203 1 GOTO
201 2 LOADMILL
202 2 LOADLATHE
203 2 PICK
203 3 PLACE
203 4 DOOR

3.5 EXPERIMENT FILE

In SIMAN experiment frame, all the variables, attributes, resources, queues, transporters and station names must be declared. To interface with FLEXIS, the experimental file must have these five global variables:

VARIABLES: 1, Source:
           2, Msgid:
           3, Dest:
           4, Numparams:
           5, Sig;

3.6 EXAMPLE

Consider a system where a ROBOT sends a message to the MILL machine to process the part, after the ROBOT loads the part into the machine. Here we can see how a ROBOT sends the REQUEST message to load the MILL and waits for a REPLY. The MILL machine receives the messages and sends back replies. In the system and configuration files, it is declared that the actor IDs of MILL and ROBOT are declared as 201 and 203, and the Msgid of the RUN is declared as 1.

ROBOT MODULE

Simulation program.
Assign: 
  Source=203:  
  Dest=201:  
  Msgid=1:  
  Sig=1:  
  Numparams = 1:  

; Since Numparams value is 1. One more variable is needed to ; send the message. Delay is declared in the experiment file.  

Delay = 10;  

; This is used for delay time in MILL to process the part  

EVENT,  2;  

; Uses event 2 to send the REQUEST.  

Queue,  Wait1q;  
Wait,  1;  

; Waits Signal when REPLY comes.  

MILL MODULE  

LOAD Queue, MillQ;  
Seize: MILL;  
Delay: A(1);  
Release: MILL;  

; Mill module is ready to give the REPLY message to ROBOT ; module.  

Assign:  
  Source=201:  
  Dest=203:  
  Msgid=1:  
  Numparams = 0;  

EVENT,  3: DISPOSE;  

; Sends the REPLY and is disposed.
CHAPTER IV
IMPLEMENTATION OF SIMAN INTERFACE

The SIMAN language allows the user to include lower-level-language functions and routines in the simulation [10]. When modeling complex systems it is not possible to capture all the systems logic using the SIMAN blocks and elements. The user code option provides the flexibility to develop complex discrete event models.

User code can be entered from the SIMAN in a variety of ways. In most of the cases the entity travelling through the model calls the user code. Typical examples would be when an entity requires a decision by a user function, or when an entity enters an EVENT block. User code can be applied to discrete event modeling. A discrete-event model is constructed by defining the events during which the system state changes and by then modeling the logic associated with each event. Events can either reschedule themselves or schedule other events to occur. Because time doesn't advance within an event, changes in the system behavior can only occur at event times. The system behavior is simulated by state changes that occur as events occur.

This chapter describes the C code used to interface with the control software. This chapter describes the structures used, synchronizing with the real time, SIMAN user routines and other routines used to communicate with the control software.

4.1 IMPORTANT STRUCTURES

In order to store the message content the data structure msg is used. The elements
in the structure msg are

typedef struct msg{
    int state;
    int tn;
    int entity;
    int sig;
    int source;
    char label[10];
}MSG;

where:

state This is used to describe the state of the REQUEST and REPLY message.
When a message is arrived the value of the state variable is changed.
Different types of states are:

REQ_ARRIVED The value of the REQ_ARRIVED is 3. When a REQUEST arrives, the value of the state variable is changed to REQ_ARRIVED. The value of the state variable is changed to REQ_CLEAR when sending a REPLY.

REQ_SENT The value of the REQ_SENT is 2. When SIMAN sends a REQUEST, the value of the state variable is changed to REQ_SENT. The value of the state variable is changed to REQ_CLEAR when a REPLY comes.

REQ_CLEAR The value of the REQ_CLEAR is 1. At the beginning of the simulation, the variable state is initialized to REQ_CLEAR. Whenever SIMAN
sends a REQUEST, it checks the variable state for REQ_CLEAR and changes to REQ_SENT. When SIMAN sends a REPLY, the state variable is changed from REQ_ARRIVED to REQ_CLEAR.

*tn* The variable *tn* is used to store the transaction number of the incoming and outgoing REQUEST message. When a REPLY arrives, the transaction number is used to locate the entity so that message parameters can be assigned to its attributes. When sending a REPLY, the transaction number of the REQUEST is sent to the destination actor.

*entity* The variable *entity* is used to store the ID of the entity when SIMAN sends the REQUEST message to other actors. When a REPLY arrives, the entity ID is used to locate the entity so that message parameters can be assigned to its attributes.

*sig* The variable *sig* is used to store the value of the SIGNAL number to be sent to model file when a REPLY arrives.

*source* The source ID of the message is stored in the variable *source*. The source ID is used while sending a REPLY or while receiving a REPLY.

*label*[10] The *label* that the entity should be sent when a message arrives. These are read in from the configuration file.

The structure *actor* has two elements.

```c
struct actor{
    MSG inp_msg[MAX_INPUT_MSG];
}```
MSG  req_msg[MAX_REQ_MSG];

where,

inp_msg[MAX_INPUT_MSG]  This array is used to store information for incoming
messages. The Msgid is used as an index into this
array. The information is needed when a REQUEST
is returned.

req_msg[MAX_REQ_MSG]  This array is used to store the outgoing REQUEST
messages. This information is needed when
receiving a REPLY. The tn serves as an index into
this array.

The actual instance of the data structure actor is declared by

struct actor *Actor;

where

Actor  is a pointer to a data structure of type actor.

The important global variables used are

Var[INDEX]  The array Var[INDEX] contains the index to the global variables
(Numparams, Source, Dest, ...) declared in experiment file. The
value of the INDEX is 4. When SIMAN sends a message,
information must be retrieved from the SIMAN global variables.
The array Var is used to get the global variable number. The
number is used as a input argument in the SIMAN routine v().
Once retrieved the values of the global variables is stored in array
Value. The index to the global variable are initialized in the
Var at the beginning of the simulation using the SIMAN routine
nsym1 in SIMAN PRIME routine.

Value[INDEX] The array Value[INDEX] contains the values of the global
variables used in model file. The values of the Msgid, Source
and Dest are stored in Value[MSGID], Value[SOURCE],
Value[DEST]. The value of the number corresponding to these
variables are stored in the Var[MSGID], Var[DEST],
Var[SOURCE].

4.2 COMMUNICATION ROUTINES

In order to communicate with server, the SIMAN client has to register with the
server. The SIMAN client registers with the server with the establish routine. The
client reads the socket with a readmesg routine and writes to the socket with the
writemesg routine. These three routines are the low level routines used to
communicate with the server. These routines are located in Register.h. In order two
utilize these routines the following files should be included.

Include files

1) sys/types.h
2) sys/socket.h
3) netinet/in.h
4) netdb.h
5) fcntl.h
6) errno.h
7) ./inet.h
8) ./common.h
9) simlib.h
4.2.1 Routine writemesg

Synopsis
status = writemesg(dest, source, type, buf);

Parameters
char *buf;
int dest;
int source;
int type;

Returns
1 if writes message to the buffer.
0 if error.

Example
status = writemesg(dest, source, type, buf);
if (status == 0)
{
    printf("error in write\n")
    sumry() /* to generate SIMAN summary report */
    exit(1);
}

Description
This routine first checks whether the connection between the server and
SIMAN is ready using routine check, then it arranges the message in the
required format (described in Chapter 2) and writes it to the socket.

writemesg(dest, source, type, bufptr)
int dest, source, type;
char *buf;
{
    char message[TOTALMSGLEN]
    extern int Sock
    int length, w

    check(Sock, CONNEC); /*check for socket connection */
    length = strlen(buf) + HEADERLEN;
    message = sprintf(message, "%4d%4d%4d%4d%s", length,
                        dest, source, type, buf);
    if ((w = write(Sock, message, length)) < 0)
    {
        perror("sending message");
        return(0);
    }
    return(w);
}
On success this routine returns 1 or if the routine fails it returns 0.

### 4.2.2 Routine readmesg

**Synopsis**

\[
\text{re = readmesg(length,type,bufptr);}
\]

**Parameters**

- int *length;
- int *type;
- char *bufptr

**Returns**

- 1 if success
- 0 if error.

**Example**

\[
\text{re = readmesg(length,type,buf);} \quad \text{if (re = = 0)} \quad \{
\text{printf(\"error in readmesg\")}
\text{sumry();} \quad /* \text{to generate SIMAN summary report */}
\text{exit(1);} \quad
\}
\]

**Description**

This routine checks whether the connection between the server and SIMAN is ready. Next it checks whether there is any data in the socket using routine check. If there is a data, it reads the socket and stores the data in the respective parameters.

```c
int readmesg(length, type, bufptr);
int *length;
int *type;
char *bufptr
{
    extern int Value[INDEX];
    extern int Sock;
    int re = 1;
    char message[HEADERLEN+1];
    extern int Sock;

    if (check(Sock, CONNEC)) {
        if (check(Sock, ANYDATA) {
            if((re =read(Sock, message, HEADERLEN)) <0){
                perror("reading socket");
                return(-1);
            }
        }
    }
```

```c
} sscanf(message, "%4d %4d %4d %4d", length,
   Value[DEST], Value[SOURCE], type);
if((re = read(Sock, bufptr, *length-
   HEADERLEN))<0){
    perror( "reading socket");
    return(-1);
}
if (*type== CLOSE || *type== KILL) {
    close(Sock);
    exit(0)
}
return(re)
} else
return(0)
```

On success this routine returns greater than 0 or if the routine fails it returns 0.

### 4.2.3 Routine Establish

**Synopsis**

```c
establish(hostnamenoofids,idbuf,gen)
```

**Parameters**

- int noofids;
- int gen;
- char *idbuf;
- char *hostname;

**Returns**

- 1 if establishes the connection to server.
- 0 if error.

**Example**

```c
if (!establish("alpha",3,idbuf,5004)) {
    exit(0)
}
```

**Description**
The SIMAN processor connects to the server using the routine establish for connection. The input parameters to the establish routine are: the generic socket ID, host of the machine where server is started, number of actor IDs and the actor ID string. The establish routine establishes the connection with the server. It connects to the
generic port on the server using generic socket ID and host of the machine where server is started. The server then opens a unique port and sends the port number back to SIMAN. Routine establish reads this port number, closes the generic socket and connects to the unique port. When connection is made, this routine sends the actor IDs. It then waits for the GO message from the server. After receiving the GO messages for all actors the routine is exited. The establish routine returns 1, if success else it returns 0.

```c
establish(hostname, noofids, idbuf, gen)
int noofids;
int gen;
char *idbuf;
char *hostname;
{
    int gen, re, port, sockd;
    struct hostent *hp; *gethostbyname();
    struct sockaddr_in name;
    struct sockaddr_in server;
    char bufptr[TOTALMSGLEN];
    int length, dest, source, type
    sockd=socket(AF_INET, SOCK_STREAM, 0);
    hp=gethostbyname(hostname);
    bcopy((char *) hp->h_addr, (char *) &name.sin_addr, hp->h_length);
    name.sin_family=AF_INET;
    name.sin_port=htons(gen);
    if(connect(sockd, (struct sockaddr *)&name, sizeof(name)) < 0){
        return(0);
    }
    if (read(sockd, bufptr, 80) > 0)
        return(0);
    close(sockd);
    sscanf(bufptr, "%d", &port);
    Sock= socket(AF_INET, SOCK_STREAM, 0);
    hp=gethostbyname(hostname);
    bcopy((char *)hp->h_addr, (char*) &server.sin_addr,
         hp->h_length);
    server.sin_family=AF_INET;
```
server.sin_port=htons(port);
if(connec(sock,(struct sockaddr *)&name,sizeof(name))<0)
    return(0);
if(write(sockd, idbuf, strlen(idbuf)) > 0)
    return(0);
while(i<=noofids){
    readmesg(&length,&type,bufptr)
    if(type==GO)
        i++;
    else
        return(0);
}
return(1)

4.3 USER ROUTINES

In order to process the messages, the SIMAN interface program uses the several routines to scan the input message, to get the number of IDs, to assign the message content to the SIMAN entity as attributes. These routines are located in the file cimlab.lib. Routine scanbuf decodes the message parameters and assigns the message parameters to the attributes of the entity. Routine put_param is called from SIMAN CEVENT routine to store the entity attributes into the message parameters. Routine variable_val get the values of the global variables. To use these routines the following files should be included

Include Files  cimlab.lib

4.3.1 Routine scan_buf

Synopsis  scan_buf(entity,buf)

Parameters  char buf[HEADERLEN]
            int entity;

Returns  none
Example

create(&entity);
scan_buf(entity,param);

Description

This routine decodes the contents of the input message character string. This routine reads the first parameter of the input character string and stores the remaining information in the another character string newbuf. Then it stores this value as an entity attribute. This process is continued until the information in the input character string is decoded.

```c
int scan_buf(entity,buf)
smint entity;
char buf[HEADERLEN];
{
    char newbuf[HEADERLEN];
    int n, j;
    float x;
    j=1;
    *newbuf = ' /0 ';
    n=sscanf(buf, " %e [\l-\177]\",&x, newbuf);
    while(n>0){
        seta(&entity,&j,&x);
        strcpy(buf,newbuf);
        newbuf= ' \0 ';
        n=sscanf(buf, " %e [\l-\177]\", &x, newbuf);
        j++;
    }
    return
}
```

4.3.2 Routine  getnoofids

Synopsis

getnoofids(idbuf);

Parameters

char  idbuf[TOTALMSGLEN]

Returns

number of ids

Example

noofids = getnoofids(idbuf);
**Description** When establishing the connection with the server, the SIMAN must supply the actor IDs and the number of IDs to the server. The SIMAN actor IDs should be stored in a format `%4d %4d %4d %4d...`. This routine gets the number of actor IDs from the configuration file and forms a string of the actor IDs in the required format. This routine also returns the number of IDs. The routine reads the first number of the input character string and stores the remaining information in the new character string newbuf. Next it stores this value in properly formatted in another character string buffer2. Now it enters a loop where it copies newbuf into input character string idbuf and buffer2 to buffer1. The next constant number is read from the input character string and the rest of the input character string is copied to newbuf. The constant number is placed into buffer2. This process is continued until all the actor IDs are stored in the required format.

```c
int getnoofids(idbuf)
    char idbuf[TOTALMSGLEN]
{
    char buffer1[TOTALMSGLEN];
    char buffer2[TOTALMSGLEN];
    char newbuf[TOTALMSGLEN];
    int i, n, x;
    i = 0;
    *newbuf = '\0';
    *buffer1 = '\0';
    n = sscanf(idbuf, " %4d [\1  -\177]", &x, newbuf);
    sprintf(buffer1,"%4d",x);
    strcpy(buffer2,buffer1);
    while(n>0){
        strcpy(idbuf,newbuf);
        strcpy(buffer1,buffer2);
        i++;
        n = sscanf(idbuf,"%4d [\1  -\177], &x, newbuf);
        sprintf(buffer2," %s %4d", buffer1,x);
    }
```
4.3.3 Routine put-param

Synopsis

put-param(entity, no_of_atrribes, param);

Parameters

int entity;
int no_of_attributes;
char param[]

Returns

char string param.

Example

put-param(entity, Value[NUMPARAMS], param);

Description

This routine is used to store the values of the attributes of an entity in the message param string. This routine writes the attributes of the entity to the string param and appends newbuf to param. It then copies param into newbuf and repeats the loop until all the attributes of an entity are stored in the param.

```
put-param(entity, no_of_atrribes, param);
smint *entity;
int no_of_attributes;
char param[];
{
    int val_attrib;
    int i;
    char new_buf[80];
    for(i=no_of_atrribes;i>l;i--)
    {
        val_attrib=a(entity,i);
        sprintf(param,"%4d %s", val_attrib, newbuf);
        strcpy(newbuf,param);
    }
    return;
}
```
4.3.4 Routine Variable_val

Synopsis

variable_val(index, Value, Var);

Parameters

int Value[];
int Var[];
int index;

Returns

none

Example

variable(4, Value, Var);

Description

This routine assigns the values of the Source, Destination, Msgid, Numparams which are stored in the SIMAN variable \( v() \) to the global array Value[]. Var[] points the proper index into \( v() \) for each of these global variables.

```c
int variable_val(index, Value, Var);
int Value[];
int Var[];
int index;
{
    int i, x;
    for(i=0; i<index; i++){
        x=Var[i];
        Value[i]=(int) v(&x);
    }
    return;
}
```

4.4 SIMAN ROUTINES

The user-written code and the SIMAN block model are interfaced when user incorporates his own routines into a specialized executable version of SIMAN. These routines must be written, compiled, and then linked to a SIMAN object and library files provided with the software. SIMAN has its own user routines, where a user can change these routines for his needs.
In SIMAN there are 10 User-Written Routines.

1) PRIME  cprime(cmn)
2) WRAPUP  cwrap(cmn)
3) UCLEAR  cuclea(cmn)
4) EVENT  cevent(Lent,Nev,cmn)
5) UF  cuf(Lent,Nuf,cmn)
6) UR  cur(Lent,Nur,cmn)
7) KEYHIT  ckeyhi(Icode,cmn)
8) USAV  curst(cmn)
9) URST  curst(cmn)
10) STATE  cstate(cmn)

The argument "cmn" is used to identify the C structure that gives the access to variables in SIMAN's FORTRAN common block SIM.

The most important SIMAN USER routines are PRIME, EVENT and WRAPUP. These three routines are used in CIMLAB for implementing the interface program. The PRIME user routine is used to register with the server and to initialize the SIMAN variables. The EVENT user routine is used to process the messages and synchronize the SIMAN simulation time with real-time. The WRAPUP routine is used to close the connection with the server and to get the summary report of the simulation results.

4.4.1. PRIME Routine

Routine PRIME is called by the SIMAN at the start of the each simulation replication. PRIME is most commonly used to initialize variables, to preload a system,
or to access data from a user file or data-base. Because this routine is called at the start of each replication, it allows the user to set or change many of the variables for the entire simulation replication. These changes can be read from a user data file or can be entered interactively through a user-written menu system.

In CIMLAB implementation, the SIMAN PRIME routine is used to register with server using TCP/IP routines and initialize the appropriate variables. The SIMAN PRIME routine open the configuration file and reads the number of SIMAN actor IDs and the labels of the model file. Next it calls the establish routine to register with the server. After SIMAN is connected, the PRIME routine uses the SIMAN NSYML routine to find the corresponding number of the global variable in the SIMAN experiment file. It stores the number of the variables in a global variable Var[], which is used in the SIMAN EVENT routine. An entity is created in the PRIME routine which is used periodically check the socket for messages. The initial simulation time and the real time is stored in the appropriate variables. The PRIME routine schedules the CEVENT routine.

```c
void cprime_( sim )
    simstr *sim;
{
    smint lent;
    smint Event_no,number;
    int index1,index2,itype,noofids;
    int len,i,j;
    int sourceid,msgida;
    int end;
    int speed;
    float velocity;
    extern float Interval,Dtime;
    extern char idbuf[TOTALMSGLEN];
    fp=fopen("./cimlab.cnf","r");
    fscanf(fp,"%4d %4d
",&min_actor,&max_actor);
    Actor=(struct actor*) malloc(( max_actor-min_actor+2) *sizeof (struct actor));
```
fscanf(fp, "%^\n", idbuf);
fscanf(fp, "%f\n", &velocity);
while (!feof(fp)) {
    fscanf(fp, "%d %d", &sourceid, &msgida);
    fscanf(fp, "%s\n", Actor [max_actor-sourceid+1].inp_msg[msgida].label);
    number++;
}
fclose(fp);
for (i=1; i<=max_actor-min_actor+1; i++) {
    for (j=0; j<MAX_REQ_MSG; j++) {
        Actor[i].inp_msg[j].state = REQ_CLEAR;
    }
}
noofids = getnoofids(idbuf);
if (!establish("alpha", noofids, idbuf, 5004)) {
    exit (0);
}
index1 = 0;
index2 = 0;
itype = 29;
len = strlen("Numparams");
Var[NUMPARAMS] = nsyml("Numparams", &len, &itype, &index1, &index2);
len = strlen("Msgid");
Var[MSGID] = nsyml("Msgid", &len, &itype, &index1, &index2);
len = strlen("Source");
Var[SOURCE] = nsyml("Source", &len, &itype, &index1, &index2);
len = strlen("Dest");
Var[DEST] = nsyml("Dest", &len, &itype, &index1, &index2);
len = strlen("VELOCITY");
speed = nsyml("VELOCITY", &len, &itype, &index1, &index2);
setv(&speed, &velocity);
create(&lent);
Dtime = 0.0;
Event_no = CHECK;
Init_sim_time = (long) sim->tnow;
Init_real_time = (long) time((time_t *) NULL);
sched(&lent, &Event_no, &Dtime);
return;

4.4.2 WRAPUP routine

Routine WRAPUP is called at the end of the each simulation replication. It is used for calculating and writing out special user statistics or the ending state of the system. It
is also used for writing the simulation results in a customized format that corresponds to
the reports obtainable from the real system.

In the CIMLAB implementation, the SIMAN WRAPUP routine sends a kill
message to the server and closes the connection with the server. The simulation is then
stopped. In WRAP routine the variables source, dest bufptr are not used. Hence they are
not initialized.

```c
void cwrap_( sim )
    simstr *sim;
{
    extern int Sock;
    int dest, source;
    char bufptr[TOTALMSGLEN];
    writemesg(dest, source, KILL, bufptr);
    close(Sock);
    exit(0);
    return;
}
```

### 4.4.3 EVENT routine

Unlike PRIME, WRAPUP, which are automatically called by SIMAN, a call to
EVENT routine must be user initiated with a entity. The most common method for calling
EVENT is to use the EVENT block or the c language syntax for the EVENT routine is
cevent(Lent, Nev, cmn)

where Lent is the record location pointer to that entity
Nev the Event number specified in the EVENT Block
   cmn is the C language structure that give access to the Fortran SIMAN routines.

When the entity arrives to an EVENT BLOCK in the model file, a call the
EVENT routine is made for that routine with the event number given in the block. The
entity pointer is passed as a first argument, Lent and the event number is passed as the
second argument. When the entity returns to the block model, the status of the entity that entered the EVENT BLOCK depends directly on what actions are specified in the user-written code. The calling entity is returned to the model file and is passed to the next block in the model. If however, the user code causes the calling entity to be disposed of or to be redirected to some other place then the calling entity is not returned. The routine SEND sends a entity to the model file. The routine SEND allows to send an entity to the block in the model file with the attached label defined in character parameter. The SIMAN SEND routine allows to define an offset time for the arrival of the entity.

In CIMLAB implementation, SIMAN EVENT routine is used for sending and receiving the messages to the server and to the model file. This is done by making an event for each type of message. There are four types of events processed by the SIMAN EVENT routine. A switch statement is used to properly direct the logic flow. The first three send a COMMAND, REQUEST and REPLY. The COMMAND event uses the routine send_cmd to send a COMMAND. The REQUEST event uses the routine send_req to send a REQUEST. The REPLY event uses the routine send_rep to send a REPLY. The fourth reads the message from the socket and process the incoming message. This event occurs at regularly spaced interval and checks whether there are any messages on the socket. If there are messages, it process the messages using routine process_msg, creates an entity and sends it to the model file. The routine process_msg is divided into three cases: process_cmd, process_req, process_rep. Each message type that comes is handled differently by the process_msg routine. Each of the above routines are explained below. After
processing the message, the simulation time is synchronized with the real time and schedules the fourth event at an interval of 1 second.

cevent(l,n,sim)
smint *1;
smint *n;
simstr *sim;
{
extern int Value[INDEX];
float dtime=0.0;
int source,type,tn,length;
int re;
int event_no=CHECK;
char bufptr[TOTALMSGLEN];
switch (*n) {
    case CMD :
        send_cmd(l);
        break;
    case REQ :
        send_req(l);
        break;
    case REPLY :
        send_rep(l);
        break;
    case CHECK :
        re = readmesg(&length,&type,bufptr);
        if (re> 1){
            printf("%s\n",bufptr);
            process_msg(Value,bufptr);
        }
        sched(l,&event_no,&dtime);
        synchronize_time(sim);
        break;
    }
    return;
}

4.5 MESSAGE ROUTINES

In order to send and receive messages, the SIMAN interface program uses the several routines. These routines are located in the file msg.lib. Routines send_cmd, send_req, send_rep sends the COMMAND, REQUEST and REPLY messages. Routines receive_cmd, receive_req, receive_rep receives the messages
from the FLEXIS toolset.

4.5.1 Routine send_cmd

Synopsis send_cmd(l)

Parameters smint *l;

Returns none

Example send_cmd(l);

Description This routine sends a COMMAND message to the server. The global variables are read using the user routine variable_val, the routine putparam puts these variables in the required format, and the message is sent to the server. Then the entity is sent back to the model.

send_cmd(l)
smint *l;
{
    extern int Value[INDEX], Var[INDEX];
    char param[TOTALMSGLEN];
    char buf[TOTALMSGLEN];
    variable_val(INDEX, Value, Var);
    put_param(l, Value[NUMPARAMS], param);
    printf("%s\n", param);
    sprintf(buf,"%4d %4d %s", CMD, Value[MSGID], param);
    writemesg(Value[DEST], Value[SOURCE], NORMAL, buf) ;
}

4.5.2 Routine send_req

Synopsis send_req(l)

Parameters smint *l;

Returns none

Example send_req(l);
Description This routine sends a REQUEST to the destination actor. The global variables are read using the routine variable_val. Next the routine searches the structure Actor to find an available entry in the REQUEST array. When it finds one with state REQCLEAR, it changes the state of the entry to REQ_SENT. Now the value of the global variable Sig and the entity ID are determined and both are stored in the REQUEST array for the use of the REPLY arrives. The index into the request array is sent as the Transaction no.. The Transaction No. in the REPLY can be used to access this information. The routine put_param is called to put the message parameters in the required format and the REQUEST message is sent.

```c
send_req(1)
smint *1;
{
    int tn,found,len,sig_no,sig;
    int itype =28;
    int index1=0;
    int index2=0;
    char param[TOTALMSGLEN],buf[TOTALMSGLEN];
    extern int Value[INDEX],Var[INDEX];
    extern int Max_actor;
    variable_val(INDEX,Value,Var);
    Value[SOURCE]=max_actor-Value[SOURCE]+1;
    for(tn=0;tn<=MAX_REQ_MSG; tn++) {
        if(Actor[Value[SOURCE]].req_msg[tn].state ==REQ_CLEAR){
            found=TRUE;
            break;
        }
    }
    Actor[Value[SOURCE]].req_msg[tn].state=REQ_SENT;
    len=strlen("Sig");
    Sig_no=nsym1("Sig",&len,&itype,&index1,&index2);
    Sig=a1(&Sig_no);
    Actor[Value[SOURCE]].req_msg[tn].sig=Sig;
    Actor[Value[SOURCE]].req_msg[tn].entity=*1;
```
4.5.3 Routine send_rep

Synopsis
send_rep(l)

Parameters
smint *l;

Returns
none

Example
send_rep(l);

Description
This routine is called to send the replies to the REQUEST message. The routine first calls the variable_val and clears the state of the input message array of the structure actor Actor. It then gets the value of the transaction number, the destination of the message from the structure Actor. It then calls the routine put_param to create the message content and sends the message to server.

send_rep(l)
smint *l;
{
    extern int Value[INDEX],Var[INDEX];
    int tn,destination;
    char param[TOTALMSGLEN];
    char buf[TOTALMSGLEN];
    variable_val(INDEX,Value,Var);
    Value[SOURCE]=max_actor-Value[SOURCE]+1;
    Actor[Value[SOURCE]].inp_msg[Value[MSGID]].state =REQ_CLEAR;
    tn=Actor[Value[SOURCE]].inp_msg[Value[MSGID]].tn;
    Dest=Actor[Value[SOURCE]].inp_msg[Value[MSGID]].source;
    put_param(l,Value[NUMPARAMS],param);
    Value[SOURCE]=max_actor-Value[SOURCE]+1;
    sprintf(buf,"%4d%4d%4d %s",REPLY,tn,Value[MSGID], param);
    writemsg(Dest,Value[SOURCE],NORMAL,buf);
}
4.5.4 Routine process_msg

Synopsis   process_msg(Value,buffer)

Parameters  int Value[];
             char buffer[TOTALMSGLEN];

Returns    none

Example    process_msg(Value,buffer);

Description  This routine scans the input buffer to determine the type of message. The
             routine process the contents of the message differently for each message
             type. A switch statement is used to direct the logic flow.

```c
int process_msg(Value,bufptr)
int Value[];
char bufptr[TOTALMSGLEN];
{
    int type;
    sscanf(bufptr,"%4d",&type);
    switch(type)  {
        case CMD :
            receive_cmd(bufptr);
            break;
        case REQ :
            receive_req(bufptr);
            break;
        case REPLY :
            receive_rep(bufptr);
            break;
    }
    return;
}
```

4.5.5 Routine receive_cmd

Synopsis   receive_cmd(buffer)

Parameters  char buffer[TOTALMSGLEN];
returns none

Example receive_cmd(buffer);

Description This routine scans the input variable bufptr to determine the Msgid and puts it in the global variable Value[MSGID]. An entity is created in the routine and parameters are scanned from the buffer using the routine scan_buf. The values of the message parameter are assign as attributes of the newly created entity. The destination of the entity is determined and it is finally sent to the appropriate block in the model file.

receive_cmd(bufptr)
char bufptr[TOTALMSGLEN];
{
    extern int lenta;
    extern int Value[INDEX];
    int slen,type;
    float dtime=0.0;
    char param[TOTALMSGLEN];
    sscanf(bufptr,"%4d %4d %s",&type,&Value[MSGID],param);
    create(&lenta);
    scan_buf(lenta,param);
    Value[DEST]=max_actor-Value[DEST]+1;
    slen=strlen(Actor[Value[DEST]].inp_msg[Value[MSGID]].label);
    send(&lenta,&Dtime,Actor[Value[DEST]].inp_msg[Value[MSGID]].label,&slen);
}

4.5.6 Routine receive_req

Synopsis receive_req(buffer)

Parameters char buffer[TOTALMSGLEN];

Returns none

Example receive_req(buffer);

Description This routine scans the input variable bufptr to determine the Msgid and
puts it in the global variable *Value[MSGID]*. The routine also reads the transaction number. The *msgid* is used to locate an entry in the input message array. The state of the entry is set to `REQ_ARRIVED`. The transaction number and the source of the message are stored in the structure `Actor` to be used when a `REPLY` is sent. The destination of the entity is determined and it is sent to the appropriate block.

```c
receive_req(bufptr)
char bufptr[TOTALMSGLEN];
{
    extern int lenta;
    extern int Value[INDEX];
    extern int Max_actor;
    int slen, type, tn;
    float dtime=0.0;
    char param[TOTALMSGLEN];
    sscanf(bufptr, "%d%d%d%s", &type, &Tn, &Value[MSGID], param);
    Value[DEST]=Max_actor-Value[DEST]+1;
    Actor[Value[DEST]].inp_msg[Value[MSGID]].state = REQ_ARRIVED;
    Actor[Value[DEST]].inp_msg[Value[MSGID]].tn=Tn;
    Actor[Value[DEST]].inp_msg[Value[MSGID]].source = Value[SOURCE];
    create(&lenta);
    scan_buf(lenta, param);
    slen=strlen(Actor[Value[DEST]].inp_msg[Value[MSGID]].label);
    send(&lenta, &dtime, Actor[Value[DEST]].inp_msg[Value[MSGID]].label, &slen);
}

4.5.7 Routine receive_rep

**Synopsis** receive_rep(buffer)

**Parameters** char buffer[TOTALMSGLEN];

**Returns** none

**Example** receive_rep(buffer);
**Description**  The input buffer is scanned to determine the Msgid, transaction number and source of the message. The global variables are then updated. The transaction number is used to index into the REQUEST array in Actor and the state of the entry is changed to REQ_CLEAR. The entity ID is used to locate the waiting entity attributes. Finally, the proper signal is sent to the model file to release the entity.

```c
receive_rep(bufptr);
char bufptr[TOTALMSGLEN];
{
    extern int lenta;
    extern int Value[INDEX];
    extern int Max_actor;
    int ent_release = 1;
    int go_signal, tn.type;
    char param[TOTALMSGLEN];
    sscanf(bufptr,"%4d%4d%4d%s", &type, &tn, &Value[MSGID], param);
    lenta=Actor[Source].req_msg[tn].entity;
    scan_buf(lenta, param);
    Actor[Value[DEST]].req_msg[tn].state=REQ_CLEAR;
    ent_release=1;
    go_signal=Actor[Value[DEST]].req_msg[tn].sig;
        signal(&go_signal,&ent_release);
}
```

4.5.8 Routine synchronize_time

**Synopsis**  synchronize_time(sim)

**Parameters**  simstr *sim;

**Returns**  none

**Example**  synchronize_time(sim);

**Description**  The simulation time times moves very fast when compared to the real-time. To achieve a real-time affect, the simulation model time has to synchronize with the real-time. This is achieved by stopping the
simulation time whenever simulation time is ahead of the real-time. The
initial simulation time and the real time of the system is noted before the
entity goes to the SIMAN event routine. In this routine the present time
and the present simulation are compared. If the difference between the
simulation time is ahead of the real time, simulation process is halted until
real-time catches up.

```c
int synchronize_time(sim)
    simstr *sim;
{
    unsigned diff_time;
    long diff_sim_time;
    int time_delay_on = 1;

    diff_sim_time=(long)sim->tnow - Init_sim_time;
    if(time_delay_on){
        while(((long)time((time_t *)NULL) - Init_real_time)<=
            diff_sim_time)(
            diff_time=((long)time((time_t *)NULL)-
                Init_real_time) - (diff_sim_time);
            usleep(diff_time);
        }
    }

    if(((long)time((time_t *)NULL) - Init_real_time)> =
        diff_sim_time+MAX_OFFSET)(
        printf("\nERROR\n");
        exit(0);
    }
}
```

### 4.6 Compiling

To compile the SIMAN interface program, the following files are needed:

```c
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <time.h>
#include "/usr/local/cinema1.3.1/lib/simlib.h"
#include "/../include/cimlab.h"
#include "/../include/inet.h"
#include "/../include/register.lib"
```
The include files that are written for CIMLAB project are placed in the include directory.

`stdlib.h` The library `stdlib.h` header file contains the declarations of general utilities. The functions are used in the include files and in the SIMAN user routines.

`stdio.h` The `stdio.h` header file contains declarations for functions for input and output operations. These functions are used in the include files and in the SIMAN user routines.

`ctype.h` The `ctype.h` header file has macros for testing and mapping characters. These functions are used in SIMAN user routines and in all the include files.

`time.h` The `time.h` contains the functions required for the timer operations. These functions are used in `msg.lib` to synchronize the simulation time with real time.

`simlib.h` `simlib.h` contains some definitions and the declarations necessary to call SIMAN FORTRAN routines.

`cimlab.h` The file `cimlab.h` contains all the declarations, structure definitions used by the C interface to SIMAN. These declarations are used in SIMAN user routines and all library files.

`inet.h` The file `inet.h` contains all the declarations necessary for the registration of SIMAN with server. These declarations are used in
register.lib library file.

msg.lib  The library file msg.lib contains all the routines to process the incoming and outgoing messages. This file also has the routine that synchronizes the simulation time with real time.

cimlab.lib  This library file cimlab.lib contains all the user routines. This routines are used to process the messages. The routines in the message.lib uses these routines.

register.lib  This library file consists functions that register with the server, write the messages to the socket and reads the messages from the socket.

cimlab.cnf  This is the configuration file. This file consists all the SIMAN actors IDs and Msgid. The SIMAN PRIME routine uses this file to read the actor IDs, message IDs and labels.

The SIMAN userc.c file is located in /user/local/cinemal.03/lib directory. This file consists all templates for the SIMAN user routines. The user can change these routines for his needs. The file should be copied into the home directory. For CIMLAB project, this file is copied and named as cimlab.c. The PRIME routine, WRAP routine and EVENT routines are changed in the cimlab.c. To compile the cimlab.c file the following command is given at the prompt:

> complink cimlab

The above command creates an executable file cimlab and saves it in the current directory.
CHAPTER V

THE CIMLAB

The SIMAN interface program is implemented in CIMLAB, a flexible manufacturing system. The simulation and control software implemented in the CIMLAB are UNIX based. The control software tool, FLEXIS, is successfully interfaced with the SIMAN/CINEMA simulation software.

5.1 CIMLAB DESCRIPTION

CIMLAB contains a Emco VMC-100 CNC mill, EMCOTURN 120P CNC lathe, GMFanuc S-10 robot, feeders and bin.

5.1.1 CNC Machines

The EMCO VMC-100 milling machine is operated with Emcotronic TM02 CNC controller, which has a control panel with VDU display and RS232 serial port to load and execute the programs. The EMCOTURN 120P turning machine is operated with Emcotronic TM02 CNC controller, which has a control panel with VDU display and RS232 serial port to load and execute the programs.

5.1.2 Gm Fanuc's S-10 Robot

The S-10 robot mechanical unit is an articulated six-axis robot powered by brushless AC servomotors.
Figure 5.1 CIMLAB Layout
Figure 5.2 Mill Machine

Figure 5.3 Lathe Machine
5.1.3 Feeders & Bin

The feeders are units to present a part to be picked by the robot. The bin is used by ROBOT to deposit the finished parts.

Figure 5.4 Gm Fanuc S-10 Robot

5.1.4 Computer

The lab is equipped with two Sun Sparc station II ("alpha" and "phoenix") and three IBM compatible computers. The Sun Sparc workstation II ("alpha") is the cell...
controller. This workstation acts as a controller to CIMLAB. FLEXIS toolset is used to develop the cell controller. The Sun Sparc workstation II ("phoenix") is used to simulate and animate the CIMLAB. The software used to simulate and animate is SIMAN/CINEMA. Two IBM PCs ("gamma" and "beta") are interfaced to the CNC machines. They use a digital I/O port to control the machine tool unit and a RS232 port to communicate with CNC controller. The third computer ("delta") is used to communicate with the robot by using two RS232 ports. One port is used for file operations and the other port is used for off-line programming. The complete documentation of the CIMLAB and PC programs is described in the report "Cimlab Server, PC programs and CNC Machines" documented by Mr. Senthil Kumar. M.

5.2 IMPLEMENTATION

A SIMAN simulation program has been successfully written for the CIMLAB. For every component of the CIMLAB, a simulation program is developed in SIMAN. The controller (developed in FLEXIS toolset) sends the messages simultaneously to both SIMAN and the physical machines. In CINEMA animation, the Robot takes a part from the FEEDER and places the part in the machines. These machines process the part. The Robot removes the part and places the finished part in the BIN. The control logic for the Robot and the machines is handled by the CELL controller. Whenever the SIMAN receives a message from the controller, it simulates the process and replies back to the controller just as the physical machines do. When the controller sends the Kill message to SIMAN model, the simulation model exits and gives the simulation summary report to the user.
CHAPTER VI

DESCRIPTION OF SIMULATION MODEL

Practical application of simulation involves very large and complex systems. It is convenient to approach the modeling of large systems by dividing them into several subsystems. Large systems have typically natural boundaries that suggest a systematic segmentation approach. For example, a large manufacturing system usually comprises a set of distinct workstations or workcells. A natural approach to modeling these types of systems is to develop models of each individual workstation and workcell. These models can then be combined to represent the overall system. A model in a simulation model represents a workcell or a department of a larger facility.

This chapter discusses the SIMAN model of the CIMLAB, where the SIMAN interface program is implemented, and about the modelling of robot model and CNC machines in the CIMLAB.

6.1 Description of the CIMLAB

The complete details of the CIMLAB are described in Chapter 5. In order to develop the simulation model for this laboratory, the SIMAN model is divided into models. These models are again divided into methods according to the type of message received by the model. A method is a simulation program in which it performs a particular set of actions. The simulation is designed to respond to the FLEXIS control software toolset and behaves the same as the actual physical system (Figure 6.1). The
FLEXIS software sends and receives the messages from SIMAN and physical models.

6.2 ROBOT

The ROBOT model is designed to simulate the actual GM-Fanuc S-10 robot. The ROBOT model is designed to respond to control software tool FLEXIS.

6.2.1 ROBOT Behavior

The GM-Fanuc S-10 robot performs three types of actions. The physical ROBOT
can place a part, pick a part or move from one position to another position. In CIMLAB, the physical ROBOT is controlled by the control software tool, FLEXIS, which sends the messages to the robot to perform the specific action. The physical robot sends the REPLY back to the FLEXIS controller.

6.2.1.1 GOTO message

When the message, GOTO arrives, the robot moves from the current position to the specified destination in the message content. There are 8 positions (Figure 6.2) where the robot can move. The robot moves with or without a part from one position to another. The CNC machine doors are controlled by the robot. The robot sends the REPLY message back to the FLEXIS.

![Figure 6.2 Positions in CIMLAB](image)

6.2.1.2 PLACE message

When a message PLACE arrives, the robot places the part it is carrying in the
machines or in the bin, depending on its position. This message arrives when the ROBOT is carrying the part and must be waiting in the positions 1, 4, 5 or 8. The robot sends the REPLY message back to the FLEXIS after placing the part.

6.2.1.3 PICK message

When a message PICK arrives, the robot picks the part from the machines or from the feeder depending on its position. This message arrives when the ROBOT is not carrying the part and must be waiting in the positions 1, 3, 6 or 8. The robot sends the REPLY message back to the FLEXIS, after picking the part.

6.1.2 SIMAN Model

The SIMAN model of the robot is designed in such a way that the model shows where the real robot is moving in the CIMLAB. This model receives messages like GOTO, PICK and PLACE from the FLEXIS Toolset (Figure 6.3). When the CELL controller sends a message to the Robot model and to the physical robot simultaneously the simulation model must simulate and animate the motion.

![Diagram](image)

**Figure 6.3** Messages in Robot
6.2.2.1 SIMAN Transporter

In SIMAN, the generic term "transporter" refers to one or more identical transfer devices that can be allocated to an entity for transferring entities between stations. In SIMAN, Transfer block provides a mechanism for transferring entities between stations. The CIMLAB employs a free-path transporter. The transporter can freely move about the system without encountering any obstructions that would delay its progress. For these free path transporters, the time to travel between stations depends on the fixed distance traveled and the speed of the transporter. In SIMAN, the transfer Block, a multi-function block, models all movements between the stations. There are three basic transfer Blocks in the SIMAN:

1) TRANSPORT
2) MOVE
3) REQUEST

The transfer block MOVE moves the transporter from the current station to the destination station. The transfer block TRANSPORT moves the transporter with the entity to the entity destination station. The REQUEST block assigns the entity to the transporter and moves the transporter to the entity's current location. In ROBOT model, the combination of REQUEST block and TRANSPORT block is used.

The REQUEST block has a problem. When a transporter is requested by an entity, the transporter should be idle; otherwise the entity that requests the transporter will enter into the queue and wait for the transporter to become idle. A transporter is said to be busy when it has a entity in it. This is a problem when ROBOT is carrying a part
with it and waiting for the GOTO message to move to different position; only the entity that is being carried by the ROBOT can instruct it to move to a different position. In order to eliminate this problem, it is compulsory to know the state of the transporter before requesting, and a virtual robot model, real and message entity are introduced.

When the robot is idle in the simulation model, the message entity transports the empty ROBOT to the required station. When the transporter is busy, the message entity bypasses the REQUEST block and sends a signal to the entity carried by the transporter, which moves the transporter to the required station. The message entity waits in a QUEUE block until the transporter reaches the destination, and then it generates the REPLY and is disposed.

6.2.2.2 Virtual Model

A virtual model is the one that is similar to the real model. The distance between the virtual model and the corresponding real model is zero, and the distance between one virtual model and any other virtual model or real model in the system is the same as the distance between the one real model and any other real model. As the distances are the same the time taken by the robot will be the same, and simulation results will not change.

6.2.2.3 Physical and Message entity

The entity that is allocated to the transporter and actually stays in the model is termed a physical entity. The entity that comes into the model file with the message content is called message entity. In other words, the model of the physical part in the real world is the physical entity. The physical entity exits the model once the part is
processed. The physical entity gets the signal from the message entity to move in the simulation model. The message-entity is created by cevent routine, when the MOVE message arrives and is disposed after the MOVE is completed.

6.2.2.4 GOTO Message

The SIMAN ROBOT model should behave in the same way as the GM-Fanuc S-10 robot in the CIMLAB. The ROBOT should move from one position to another, with or without a part. In the beginning of the simulation, the real entity is created in the robot model and waits in the feeder queue. When the GOTO message comes, the message entity is directed depending on the state of the transporter. If the ROBOT is idle, the message entity requests the transporter and moves the transporter to the destination station ID stored in its attribute A(1). After reaching the destination, the message entity sets the global variable LOAD to the value of the station number of the real station. The message entity then sends the REPLY, as described in the Chapter 3. This represents the robot moving from one position to another position without a part.

When the transporter is busy, the message entity sets the value of the global variable destination equal to the value stored in the attribute A(1) and sends the signal to the real entity that is waiting in a QUEUE block. The message entity then enters a queue and waits for the physical entity to signal when the move is complete. When the physical entity gets the signal from the message entity, it uses the global variable destination to transports the transporter to the required destination. When the robot reaches the destination station, the physical entity sends the signal to the waiting message entity, which then sends the REPLY, as described in the Chapter 3.
Go_Robot Station, Branch,1:
  IF, NT(1), Robot_busy:
  ELSE, Robot_idle;

Robot_idle Assign:
  Destination=a(1)+8;
; Finds corresponding Virtual Station for the real station.
  Queue, REQUEST:
  Siman_Robot, Velocity;
  Transport:
  Siman_Robot, Destination;

; Sends the entity to virtual station.
Station, Vfeeder;
; Enters the virtual station as robot is idle
; Sends the REPLY back.
  Assign:
    Load=4:
    SOURCE=203;
    MSGID=1
    NUMPARAMS=0
  Free:
    Siman_Robot;
; The Robot becomes idle.
Robot_busy Assign:
  Destination=a(1);
; Assigns the attribute value to the variable
  Signal:
    25;
; Sends signal to the physical entity to transport the robot
  Assign:
    Reached=10;
  Queue, REQUEST:
    Robot_BusyQ;
  Wait:
    Reached;
; Waits for the signal from physical entity.
  Event, 3:Dispose;
; When signal arrives it sends the REPLY
  Assign:
    Load=4:
    SOURCE=203;
    MSGID=1
    NUMPARAMS=0
  Event, 3:Dispose;
Queue, Wait1q; 25;
; physical entity waiting for the signal from message entity

Transport, Siman_Robot,mill;  
Station, MILL  
Signal, Reached;

; signal is given to the message entity.

Queue, Wait2q; load;
; real entity waiting for the signal to load the part.

STATION, Mill;  
SIGNAL: Reached;

; Sends signal to message entity

ASSIGN: Unload=8;  
QUEUE, MLoad_In_WaitQ;  
WAIT: 8; Waits to load the part.

; Gets the SIGNAL to Load the part.

SIGNAL: 26;
; gives the signal to message entity.

FREE: Siman_Robot;  
ASSIGN: Parttype=1:  
                  Load=8;

QUEUE, MLoad_Out_WaitQ;  
WAIT: 8;  
SIGNAL: 27;  
ASSIGN: Load=25;  
QUEUE, MUnload_Out_WaitQ;  
WAIT: 25;

; Moves to Real Station

TRANSPORT: Siman_Robot,Destination;

6.2.2.5 PICK Message

When the message PICK arrives, the message entity enters the LOAD method and
gives the signal to the physical entity to load the part in the machine. It then enters the QUEUE block and waits for the part to be loaded in the machine. When the physical entity receives the signal LOAD from the message entity, the physical entity moves the transporter to the proper location and loads the part into the machine.

After the part is loaded, the physical entity sends a signal to message entity and enters a QUEUE block. When the message entity receives the signal, it sends the REPLY, as described in the Chapter 3.

```
Load_Rob      Signal,  Load;
             Queue,   waitq;
             Wait:    Load_reached;

; Waits for the signal from real-entity
; sends the message to SIMAN interface program

Assign:      SOURCE=203;
             MSGID=1
             NUMPARAMS=0

Event,       3:DISPOSE

; real entity waits in the queue for the signal from message entity

Queue,       Wait1q;
             Load;

; gets the signal and requests the robot to load the part.

Queue        Feederq;
REQUEST,     Siman_Robot,Velocity;
Transport,   Siman_Robot,Feeder;

; takes a part from the feeder and simulation shows the robot as busy whenever it is carrying a part
; it gives the signal to message entity and waits for another message

Station,     Feeder
Signal,      Load_reached;
Queue,       Wait2q;
```
6.2.2.6 PLACE message

When the message PLACE arrives, the message entity enters the UNLOAD method, gives the signal to the physical entity to unload the part and waits in a QUEUE block for the part to be unloaded. When the physical entity receives the signal UNLOAD from the message entity, the physical entity moves the transporter and unloads the part. After unloading the part, the physical entity sends a signal to the message entity and enters a QUEUE block. The message entity upon receiving the signal, sends the REPLY, as described in the Chapter 3.

; message entity enters and signals the real-entity to unload
unload_Rob  Signal, unload
Assign: unload_reached=10;

; message entity waits for the signal from real-entity
Queue, waitq;
Wait: unload_reached;

; message is passed to SIMAN interface program
Assign: SOURCE=203;
      MSGID=1
      NUMPARAMS=0
Event, 3;

; real entity waits for the signal from message entity
Queue, Waitlq;
Wait, unload;

; transports the entity to the bin
Transport, Siman_Robot,bin;

; Robot is freed and real entity sends the signal to
6.3 CNC MACHINES

The machine models in the CIMLAB simulation perform the actions like loading the CNC program and executing the program to process the part. The machine model responses to the messages from the control software (Figure 6.4). The machine model must behave similar to the physical machines.

![Diagram of Machine Model](image)

**Figure 6.4 Messages in Machine Model**

### 6.3.1 CNC MACHINE Behavior

The CNC machines perform two types of actions. They are:

1) LOAD

2) RUN.

#### 6.3.1.1 LOAD

When the message LOAD arrives, a CNC program is loaded into the machine.

After loading the program, a REPLY is sent back.
6.3.1.2 RUN

When a message RUN arrives, the CNC program is executed and processes the part. After executing the part, a REPLY is sent back.

6.3.2 SIMAN MACHINE MODEL

In CIMLAB, the machine model is duplicated to represent the MILL and LATHE. The MILL and LATHE models of the CIMLAB have the same SIMAN functions with different label names and resource names. The MILL model simulates the MILL machine and the LATHE model simulates the LATHE machine.

6.3.2.1 RUN message

In this model, the time taken for machining the part is represented by a DELAY Block. The time specified in the DELAY Block is specified in the attribute A(1) of the message entity. The execution of the program is represented by seizing a resource. After processing the part in the machine, it releases the resource. It then sends the REPLY, as described in the Chapter 3.

```
Machine RUN Queue, MachineQ;
Seize: MachineRes;
Count: MachinePart;
Assign: Parttype=1;
;Assigns CINEMA attribute
Delay: A(1);
;Delays by the amount specified in A(1)
Release MachineRes;
; Sends the REPLY to the FLEXIS
```
In the above machine model, the names of the queues and resources should be different for the MILL and LATHE models, and the names used in these models should be declared in the experiment file. For animation, CINEMA can be used. To see the entity in the animation, an attribute should be declared for that entity. (In the above example model, the CINEMA attribute is \texttt{PARTTYPE})

6.3.2.2 LOAD Message

Before processing the part in the machine, a CNC program is loaded into the machine. In SIMAN, the loading of a program can be simulated by seizing a machine for the time taken to load the program in the CNC machine. In this model, the time taken for loading the program in the CNC machine is represented by a \texttt{DELAY} Block. The time specified in the \texttt{DELAY} Block is specified in the attribute \texttt{A(1)} of the message entity. When the program is loaded, the message entity releases the machine and sends the \texttt{REPLY} back, as described in Chapter 3.

```
LOADMACHINE
  Queue, Programmemachineq;
  Seize: Programmemachine;
  Assign: Parttype=1;
  Delay: A(1); Delays the time ProgrammeMachine;
  Release
;
Sends the \texttt{REPLY} back to the FLEXIS
```

```
Assign: SOURCE=201:
MSGID=2:
```
Event: 3:
Dispose;

6.4 CIMLAB EXPERIMENT FILE

In SIMAN Experiment Frame, all the variables, attributes, resources, queues, transporters and station names should be declared. The experiment file of the CIMLAB model is given below:

BEGIN;
PROJECT, CIMLAB,Lanka Somanath,02/14/93;
VARIABLES:
  1,SOURCE:
  2,MSGID:
  3,DEST:
  4,NUMPARAMS:
  5,velocity:
  6,entitylocation,7:
  7,Destination:
  8,go:
  9,unload:
 10,load:
 11,reached:
 12,unload_reached:
 13,load_reached;
ATTRIBUTES:
  1,parttype;
  2,Robot_IdleQ:
  3,Robot_BusyQ:
  4,Robot_UnloadQ:
  5,Unload_WaitQ:
  6,Robot_LoadQ:
  7,Load_WaitQ:
  8,FeederMillQ:
  9,MFeederQ:
 10,MPartWait:
 11,FeederLatheQ:
 12,LFeederQ:
 13,LPartWait:
 14,Robot_WaitQ:
 15,MLoad_In_WaitQ:
 16,MLoad_Out_WaitQ:
 17,MUnload_In_WaitQ:
 18,MUnload_Out_WaitQ:
 19,LDoorQ:
 20,DoorQ:

NumParams=0:
RESOURCES:
1. millres:
2. latheres:
3. programmemill:
4. programmelathe:
5. Robo_loading:
6. Robo_unloading:
7. doorlathe:
8. doormill;

COUNTERS:
1. parts:
2. lathepart:
3. millpart;

STATIONS:
1. lathe:
2. dummy11:
3. lathefeeder:
4. lathebin:
5. millbin:
6. millfeeder:
7. dummy12:
8. mill:
9. vlathe:
10. vdummy11:
11. vlathefeeder:
12. vlathebin:
13. vmillbin:
14. vmillfeeder:
15. vdummy12:
16. vmill:
17. idle:
18. Mill_In_Unload:
19. Mill_Out_Unload:
20. Mill_In_Load:
21. Mill_Out_Load:
22. Lathe_In_Unload:
23. Lathe_Out_Unload:
24. Lathe_In_Load:
25. Lathe_Out_Load;

TRANSPORTERS: 1. SIMAN_ROBOT,,DISTANCE(1),,IDLE-A;
DISTANCES ELEMENT;
LAYOUTS: "cimlab.lay",parttype,0.1,0.1,"s";
6.4.1 Variables

The variables SOURCE, MSGID, DEST, NUMPARAMS are used to send the messages, and they are described in detail in Chapter 3. The variable Velocity is used for the transporter velocity. The variable Destination is used to send the transporter to the destination station. The variables Go, Unload, Load, Reached, Unload_reached, and Load_reached are used to send the signals between the real and message entities. The values of these variables depend upon the station numbers where the message entity is residing.

6.4.2 Attributes

The attribute parttype is used for cinema.

6.4.3 Resources

The resources Millres, and Latheres are used in the machine model for execution of the CNC program. The resources Doormill and Doorlathe are used in the ROBOT model to open and close the doors of the machines. The resources Robo_loading and Robo_unloading used in the ROBOT model are used for CINEMA to show the ROBOT is loading and unloading.

6.4.4 Stations

The stations in the 1 to 8 are the actual positions of the CIMLAB. The stations 9 to 16 are the corresponding virtual stations. The stations 18 to 25 are used for
CINEMA to show the ROBOT moving into the machines and coming out from the machines.

6.5 ANIMATION WITH CINEMA

CINEMA is a animation software that runs concurrently with SIMAN. Detailed animations of any SIMAN model can be constructed rapidly using CINEMA. The user sees a real-time graphical animation of the SIMAN model jobs or customers moving through the system. CINEMA builds the animation without any programming effort. Animation construction is a two step process. The first step consists of building an animation layout using CINEMA. The layout is a graphical representation of the SIMAN model. The SIMAN model and the animation layout are brought together for the real-time animation and simulation.

Cimlab.lay This file contains the layout of the CIMLAB.
Resource.res This file contains all the resources used in CIMLAB.
Entity.ent This files has the drawings of the entities.
Robot.tra The transporter used in the CIMLAB is Robot. This file has the drawings of robot used in the CIMLAB.
Cimlab.glo The global variables used in the cimlab are shown in this file. There are used to show the states of the CNC machines and the Robot in the CIMLAB.

A complete layout for demonstration purposes is created and named as CIMLAB.LAY. While linking with the simulation model, the following command should be entered in
response to the prompt:

> cimlab cimlab.p cimlab.lay

### 6.6 Compilation of Files

The mill module is compiled by entering the following command in response to the prompt:

> model millmodule.mod millmodule.m

The above command creates the file millmodule.m and saves it in the current directory.

The lathe module is compiled by entering the following command in response to the prompt:

> model lathemodule.mod lathemodule.m

The above command creates the file lathemodule.m and saves it in the current directory.

The Robot module is compiled by entering the following command in response to the prompt:

> model robotmodule.mod robotmodule.m

The above command creates the file robotmodule.m and saves it in the current directory.

In order to create the siman simulation program file, experiment file is compiled by entering the following command in response to the prompt:

> expmt cimlab.exp cimlab.e

The above command creates the file cimlab.e and saves it in the current directory.

To create the simulation program file, cimlab.p, the model program files and experiment program files are linked and compiled using the following command at the prompt:
> linker  robotmodule.m+millmodule.m+lathemodule.m  cimlab.e
   cimlab.p

The above command creates the file cimlab.p and saves it in the current directory.

To compile the SIMAN C routines or SIMAN userC file, the following command is given at the prompt:

> complink  cimlab

The above command creates an executable file complink and saves it in the current directory.
CHAPTER VII

CONCLUSIONS

A generic interface for the SIMAN simulation language was developed. This interface includes the C code required to establish and communicate with the server using the TCP/IP and sockets. A generic message protocol scheme was developed. The messages are used to exchange the information between the SIMAN and other computer processes respectively. The protocol extensions to the SIMAN were developed so that any SIMAN simulation model can send and receive messages using the message protocol. Also a method to synchronize the simulation model with real-time was coded.

In addition, a SIMAN simulation model was developed to simulate and animate the physical equipment in the CIMLAB. The model, simulates and animates the Mill CNC machine, the Lathe CNC machine and the GM-Fanuc S-10 Robot. The simulation can be controlled by the CIMLAB control system (developed by Senthil (1994) and Sathya (1993)). The simulation is animated using CINEMA (animation tool). The animation of the CIMLAB shows the state of the system at any instant of time and the animation of the physical system can be viewed from any remote machine.

The simulation model is implemented in separate objects, which are independent of each other. The objects communicate with each other using messages. The simulation models of Mill and Lathe are instances of the generic machine object class. The data encapsulation technique of the object-oriented design is used to give more flexibility for implementing other simulation models. This simulation model can be used to test and
debug the control system.

In future work, the simulation model can be improved to test the complex control logic in large manufacturing systems. A generic machine model, which includes the breakdown of the machines, can be developed. A generic transport model can be developed for conveyors and/or automatic guided vehicles.
REFERENCES


10. Shannon Pedsky *Introduction to Simulation using SIMAN*, System Modeling
Corporation, 1990.


APPENDIX A

HOW TO RUN CIMLAB

A.1 CIM LAB LOGINS

Appendix A describes how to run the CIMLAB. To develop the CIMLAB, two users, *devel* and *demo*, exist on "alpha", a Sun SPARC Station 2 networked with internet.

A.1.1 *demo* login

The *demo* login is used when the CIMLAB is used for the demo purpose. The complete file and the hierarchical structure is described in the thesis "Object Oriented Cell Controller in a Manufacturing Facility" by Gopal Reddy Sathyanarayana Reddy.

A.1.2 *devel* login

The *devel* login is used when the CIMLAB is in the development and creating of new programs. The simulation software, SIMAN, executes only on "phoenix"; another Sun SPARC Station II, so *demo* and *devel* logins are created in "phoenix" for SIMAN, and it is remotely executed from "alpha".

A.2.1 How to Run Simulation

Step 1) Click the left mouse button on **START** in the Process Execution window.

Step 2) Point mouse cursor in CINEMA window and press esc. The CINEMA window displays the menu options. Click **RUN** from the menu options.
Step 3) Observe CINEMA to see your program executing.

A.2.2 How to Exit Simulation

Step 1) If CINEMA window is running, press <esc>
Step 2) Click QUIT from menu options.
Step 3) Exit Openwindows.
Step 4) Logout from the machine.

A.3 HOW TO START CIMLAB

Shell programs are written to run different CIMLAB programs. gocim is a shell program which opens different windows for flexis, runtime, server and cinema and starts all the programs. To run individual programs; goflexrun, goserver and gosiman are used. goflexrun runs only the flexis and runtime programs, goserver runs only the server, with the options of selecting the server to run with realtime machines and/or CINEMA and gosiman runs only the CINEMA program.

The following steps describe how to start the CIM laboratory:

A.3.1 With Siman

Step 1: Start Openwindows.
Step 2: Start gocim and give N (No) to the Server options to start the Server for only Siman. (Note: gocim starts runtime also)
Step 3: Start the FLEXIS as described in Section D.3 of "Object Oriented CELL
CONTROLLER for a Manufacturing Facility" by Gopal Reddy Sathyanarayana Reddy.

Step 4: Start the CINEMA as described in section A.2.1.

Step 5: When the execution is over, Exit the CINEMA as described in section A 2.2, which also kills the Server. Exit the Flexis as described in Section D.5 of "Object Oriented CELL CONTROLLER for a Manufacturing Facility" by Gopal Reddy Sathyanarayana Reddy. Exit Runtime by typing shutdown in runtime window.

A.3.2 With Machines and Siman

Step 1: Start Openwindows.

Step 2: Start gocim and give Y (Yes) to the Server options to start the Server for Siman and RealTime Machines. Note: gocim starts runtime also.

Step 3: Start the CNC Machines, Robot and the beta, gamma and delta PC's.

Step 4: Start the Flexis as described in Section D.3 of "Object Oriented CELL CONTROLLER for a Manufacturing Facility" by Gopal Reddy Sathyanarayana Reddy.

Step 5: Start the CINEMA as described in section A 2.1.

Step 6: In PC's goto cimlab directory, run demo or devel to mount the required directories for the PC's.

Step 7: Run RUN to start the programs. Note: To simulate the RealTime Machines run sim to start simulation programs.
Step 8: When the execution is over, Exit the CINEMA as described in section 2.1, which also kills the Server. Exit Flexis as described in Section D.5 of "Object Oriented CELL CONTROLLER for a Manufacturing Facility" by Mr. Gopal Reddy Sathyanarayana Reddy. Exit Runtime by typing shutdown in runtime window.
The following files are used for simulation of the CIMLAB.

**Robotmodule.mod**
The actions performed by the robot. The robot module has several submodels and each submodel simulates only one type of actions like GOTO, PICK, and PLACE.

**Millmodule.mod**
Simulates the CNC machine in CIMLAB. The mill module has two submodels. The **mill run** submodel executes the program to process the part and the **mill load** submodel loads the program into the mill.

**Lathemodule.mod**
Simulates the CNC machine in CIMLAB. The lathe module has two submodels. The **lathe run** submodel executes the program to process the part and the **lathe load** submodel loads the program into the lathe.

**Cimlab.exp**
The global variables, resources, transporters and queues used in the CIMLAB model files.

**Cimlab.c**
Contains the c function which process the incoming and outgoing messages. This file has all the SIMAN user routines like PRIME, EVENT, and WRAP. The
cimlab.c file is compiled using the SIMAN command complink. The executable file cimlab is used to run the simulation.

**cimlab.lay**

Contains the background and complete layout of the CIMLAB. This file is used for animation of the CIMLAB. The resource file cimlab.res, the entity file cimlab.ent, the transporter file cimlab.tra and the global variable file cimlab.glo are linked to construct the layout file cimlab.lay.

**simlib.h**

Contains some definitions and the declarations necessary to call the SIMAN FORTRAN routines.

**establish.lib**

Contains the routines which are used to register with the server. This file has routines which read the messages and write the messages to the server. In order to use this file the following files are included:

- sys/types.h
- sys/socket.h
- netinet/in.h
- netdb.h
- fcntl.h
- errno.h
- ./inet.h
- ./common.h
- simlib.h

**common.h**

The common procedures that are used in server and its clients.
has the routines that check for the socket connection and uses poll. The following files are to be included to use this file

sys/time.h
sys/ioctl.h

inet.h

It has preprocessor #define directives, type definitions for error codes and message types that are used in establish.lib library file.

cimlab.cnf

The configuration file used for the simulation and animation of CIMLAB. The file consists of labels that are used to send the messages to model file and SIMAN actor IDs. These labels and actors are arranged in special format.

cimlab.lib

Has user written routines that arranges the actor IDs, ar arranges the message content in a string format. These routine is used whenever a message is processed. The include file to use this file is

simlib.h

cimlab.h

Contains the preprocessor #define directives that are used in cimlab.c
APPENDIX C

SOURCE CODE

C.1.1 cimlab.cnf

201 203
201 202 203 -1
25
201 3 MILL
201 2 LOAD_MILL
202 3 LATHE
202 2 LOAD_LATHE
203 1 GO_ROBOT
203 2 LOAD_ROB
203 3 UNLO_ROB
203 4 DOOR_OPEN
C.1.2 millmodule.mod

BEGIN;
;MILL MODULE
MILL QUEUE, MILLQ;
SEIZE: MILLres;
COUNT: Millpart;
ASSIGN: Parttype=1;
DELAY: A(1);
RELEASE: MILLres;
ASSIGN: Source=201;
eventId=3;
Numparams=0;
EVENT: 3:DISPOSE;

LOAD_MILL QUEUE, ProgrammeMILLq;
SEIZE: ProgrammeMILL;
ASSIGN: Parttype=1;
DELAY: A(1);
RELEASE: ProgrammeMILL;
ASSIGN: Source=201;
eventId=2;
Numparams=0;
EVENT: 3:DISPOSE;

END;
C.1.3 latemodule.mod

BEGIN;
; Lathe module
LATHE

| QUEUE,      | LATHEq;          |
| SEIZE:      | LATHEres;        |
| COUNT:      | Lathepart;       |
| ASSIGN:     | Parttype=1;      |
| DELAY:      | A(1);            |
| RELEASE:    | LATHEres;        |
| ASSIGN:     | Source=202;      |
|             | MsgID=3;         |
|             | Numparams=0;     |
| EVENT:      | 3: DISPOSE;      |

LOAD_LATHE

| QUEUE,      | ProgrammeLATHEq; |
| SEIZE:      | ProgrammeLATHE;  |
| ASSIGN:     | Parttype=1;      |
| DELAY:      | A(1);            |
| RELEASE:    | ProgrammeLATHE;  |
| ASSIGN:     | Source=202;      |
|             | MsgID=2;         |
|             | Numparams=0;     |
| EVENT:      | 3: DISPOSE;      |

END;
C.1.4 robotmodule.mod

BEGIN;

Go_Robot

STATION, BRANCH, 1: IDLE;

IF, NT(1). Eq. 1, Robot_busy: ELSE, Robot_idle;

Robot_idle

ASSIGN: Parttype=0;
ASSIGN: Destination= a(1)+8;
Entitylocation=LT(1, 1);

QUEUE, REQUEST:
Robot_IdleQ;
SIMAN_ROBOT, Velocity,
Entitylocation;

TRANSPORT:
SIMAN_ROBOT, Destination,
Velocity;

Robot_busy

ASSIGN: Destination= a(1):
Reached=10;
SIGNAL: 25;
QUEUE, Robot_BusyQ;
WAIT: Reached;
ASSIGN: Source=203:
Msgid=1:
Numparams=0;
EVENT: 3: DISPOSE;

STATION, VLatheFeeder;
ASSIGN: Load=11;
ASSIGN: Source=203:
Msgid=1:
Numparams=0;
FREE: SIMAN_ROBOT;
EVENT: 3: DISPOSE;

STATION, VMillFeeder;
ASSIGN: Load=14;
ASSIGN: Source=203:
Msgid=1:
Numparams=0;
FREE: SIMAN_ROBOT;
EVENT: 3: DISPOSE;

STATION, VMill;
ASSIGN: Load=8;
ASSIGN: Source=203:
Msgid=1:
FREE: SIMAN_ROBOT; 3: DISPOSE;
EVENT: 3: DISPOSE;

STATION, ASSIGN: VLathe; Load=1;
ASSIGN: Source=203; Msgid=1;
FREE: SIMAN_ROBOT; ASSIGN: Numparams=0;
EVENT: 3: DISPOSE;

Unlo_rob SIGNAL: Unload;
QUEUE, SEIZE: Robot_UnloadQ; RoBo_Unloading;
QUEUE, QUEUE, Unload_WaitQ;
WAIT: 26;
RELEASE: RoBo_Unloading;
ASSIGN: Source=203; Msgid=1;
EVENT: 3: DISPOSE;

Load_rob SIGNAL: Load;
QUEUE, SEIZE: Robot_LoadQ; RoBo_Loading;
QUEUE, QUEUE, Load_WaitQ;
WAIT: 27;
RELEASE: RoBo_Loading;
ASSIGN: Source=203; Msgid=1;
EVENT: 3: DISPOSE;

STATION, SIGNAL: MillFeeder; Reached;
BRANCH, 1: IF, nt(1).eq.1, RobotLoad:
ELSE, Mill_Part;

STATION, SIGNAL: LatheFeeder; Reached;
BRANCH, 1: IF, nt(1).eq.1, RobotLoad:
ELSE, Lathe_Part;

CREATE;
ASSIGN: Parttype=1;
MillFeeder

```
QUEUE, 
WAIT:  
BRANCH,  
ALWAYS,MillFeeder: 
ALWAYS,Mill_Part;
```

Mill_Part

```
ASSIGN:  
M = Load-8:  
EntityLocation=LT(1,1);  
QUEUE,  
REQUEST:  
SIGNAL:  
QUEUE,  
WAIT:  
TRANSPORT:  
CREATE;  
ASSIGN:  
ENTITYLOCATION;  
```

LatheFeeder

```
ASSIGN:  
Parttype=1;  
QUEUE,  
WAIT:  
BRANCH,  
ALWAYS,LatheFeeder: 
ALWAYS,Lathe_Part;
```

Lathe_Part

```
ASSIGN:  
M = Load-8:  
EntityLocation=LT(1,1);  
QUEUE,  
REQUEST:  
SIGNAL:  
QUEUE,  
WAIT:  
TRANSPORT: 
```

RobotLoad

```
ASSIGN:  
Unload=3;  
QUEUE,  
WAIT:  
ASSIGN:  
Load=4;  
SIGNAL:  
FREE:  
STATION,  
SIGNAL:  
ASSIGN:  
Unload=8;
```
QUEUE, WAIT: MLoad_In_WaitQ; 8;

TRANSPORT: Siman_Robot,Mill_In_Unload, velocity;

STATION, ASSIGN: Mill_In_Unload; Parttype=0;
TRANSPORT: Siman_Robot,Mill_Out_Unload, velocity;

STATION, ASSIGN: Mill_Out_Unload; M=8;
SIGNAL: 26;
RELEASE: DoorMill;
FREE: Siman_Robot;
ASSIGN: Parttype=1; Load=8;

QUEUE, WAIT: MLoad_Out_WaitQ; 8;
ASSIGN: EntityLocation=LT(1,1);

QUEUE, REQUEST: MUnload_In_WaitQ; SIMAN_ROBOT,Velocity;
ASSIGN: Parttype=0;
QUEUE, SEIZE: MDoorQ; DoorMill;
TRANSPORT: Siman_Robot,Mill_In_Load, velocity;

STATION, ASSIGN: Mill_In_Load; Parttype=1;
TRANSPORT: Siman_Robot,Mill_Out_Load, velocity;

STATION, ASSIGN: Mill_Out_Load; M=8;
SIGNAL: 27;
ASSIGN: Load=25;
QUEUE, WAIT: MUnload_Out_WaitQ; 25;
TRANSPORT: Siman_Robot,Destination, Velocity;

STATION, SIGNAL: Lathe; Reached;
ASSIGN: Unload=1;
QUEUE, WAIT: LLoad_In_WaitQ; 1;
TRANSPORT: Siman_Robot,Lathe_In_Unload,
velocity;
STATION, Lathe_In_Unload;
ASSIGN: Parttype=0;
TRANSPORT: Siman_Robot,Lathe_Out_Unload,
velocity;

STATION, Lathe_Out_Unload;
ASSIGN: M=1;
SIGNAL: 26;
RELEASE: DoorLathe;
FREE: Siman_Robot;
ASSIGN: Parttype=1;
Load=1;
QUEUE, LLoad_Out_WaitQ;
WAIT: 1;
ASSIGN: EntityLocation=LT(1,1);
QUEUE, LUnload_In_WaitQ; LatheRobot;
REQUEST: SIMAN_ROBOT,Velocity,m;
ASSIGN: Parttype=0;
QUEUE, LDoorQ;
SEIZE: DoorLathe;
TRANSPORT: Siman_Robot,Lathe_In_Load,
velocity;
STATION, Lathe_In_Load;
ASSIGN: Parttype=1;
TRANSPORT: Siman_Robot,Lathe_Out_Load,
velocity;

STATION, Lathe_Out_Load;
ASSIGN: M=1;
SIGNAL: 27;
ASSIGN: Load=25;
QUEUE, LUnload_Out_WaitQ;
WAIT: 25;
TRANSPORT: Siman_Robot,Destination,
Velocity;

DOOR_OPE BRANCH, 1:
IF, NR(doormill) >= 0, MillOpen:
ELSE, LatheOpen;

MillOpen QUEUE, MillDoorq;
SEIZE: DoorMill;
ASSIGN: Source=203:
Msgid=4:
Numparams=0;
EVENT: 3:DISPOSE;

LatheOpen QUEUE, LatheDoorq;
SEIZE: DoorLathe;
ASSIGN: Source=203:
      Msgid=4:
      Numparms=0:
EVENT: 3:DISPOSE;

STATION, SIGNAL: MillBin;
ASSIGN: Reached;
ASSIGN: Unload=5;
QUEUE, SIGNAL: MillBinQ;
WAIT: 5;
FREE: Siman_RObot;
SIGNAL: 26:DISPOSE;

STATION, SIGNAL: LatheBin;
ASSIGN: Reached;
ASSIGN: Unload=5;
QUEUE, SIGNAL: LatheBinQ;
WAIT: 5;
FREE: Siman_RObot;
SIGNAL: 26:DISPOSE;

END;
C.1.5 cimlab.exp

BEGIN;
PROJECT, CIMLAB, Lanka Somanath, 02/14/93;
VARIABLES: 1, SOURCE:
2, MSGID:
3, DEST:
4, NUMPARAMS:
5, velocity:
6, entitylocation, 7:
7, Destination:
8, go:
9, unload:
10, load:
11, reached:
12, unload_reached:
13, load_reached:

ATTRIBUTES: 1, Execute:
2, parttype:
3, loadstation:

QUEUES: 1, Robot_IdleQ:
2, Robot_BusyQ:
3, Robot_UnloadQ:
4, Unload_WaitQ:
5, Robot_LoadQ:
6, Load_WaitQ:
7, FeederMillQ:
8, MFeederQ:
9, MPartWait:
10, FeederLatheQ:
11, LFeederQ:
12, LPartWait:
13, Robot_WaitQ:
14, MLoad_In_WaitQ:
15, MLoad_Out_WaitQ:
16, MUnload_In_WaitQ:
17, MUnload_Out_WaitQ:
18, MDoorQ:
19, LDoorQ:
20, LLoad_In_WaitQ:
21, LUnload_Out_WaitQ:
22, LLoad_Out_WaitQ:
23, LUnload_In_WaitQ:
24, MillDoorQ:
25, LatheDoorQ:
26, MillBinQ:
27, LatheBinQ:
28, MILLQ:
29, ProgrammeMILLQ:
30, Latheq:
RESOURCES:  1, millres:
            2, latheres:
            3, programmemill:
            4, programmelathe:
            5, Robo_loading:
            6, Robo_unloading:
            7, dooralathe:
            8, doormill:

COUNTERS:  1, parts:
            2, lathepart:
            3, millpart:

STATIONS:  1, lathe:
            2, dummy11:
            3, lathefeeder:
            4, lathebin:
            5, millbin:
            6, millfeeder:
            7, dummy12:
            8, mill:
            9, vlathe:
            10, vdummy11:
            11, vlathefeeder:
            12, vlathebin:
            13, vmillbin:
            14, vmillfeeder:
            15, vdummy12:
            16, vmill:
            17, idle:
            18, Mill_In_Unload:
            19, Mill_Out_Unload:
            20, Mill_In_Load:
            21, Mill_Out_Load:
            22, Lathe_In_Unload:
            23, Lathe_Out_Unload:
            24, Lathe_In_Load:
            25, Lathe_Out_Load:

TRANSPORTERS:  1, SIMAN_ROBOT, ,DISTANCE(1), ,IDLE-A:

DISTANCES:  1, Lathe-Mill-250,   Lathe-LatheFeeder-125,
             Lathe-Idle-125,    Lathe-LatheBin-125,
             Lathe-VMill-250,   Lathe-VLatheFeeder-125,
             Lathe-MillBin-125,
             Lathe-Mill_In_Unload-250,
             Lathe-Mill_In_load-250,
             Lathe-Mill_Out_Unload-250,
             Lathe-Mill_Out_Load-250,
             Lathe-Lathe_In_Unload-25,
             Lathe-Lathe_In_load-25,
Lathe-Lathe_Out_Unload-0,
Lathe-Lathe_Out_Load-0,
Lathe-VLathe-0,
VLathe-Mill-250,    VLathe-LatheFeeder-125,
VLathe-Idle-125,    VLathe-LatheBin-125,
VLathe-VMill-250,    VLathe-VLatheFeeder-125,
VLathe-MillBin-125,
VLathe-Mill_In_Unload-250,
VLathe-Mill_In_load-250,
VLathe-Mill_Out_Unload-250,
VLathe-Mill_Out_Load-250,
VLathe-Lathe_In_Unload-25,
VLathe-Lathe_In_load-25,
VLathe-Lathe_Out_Unload-0,
VLathe-Lathe_Out_Load-0,
VLathe-Lathe-0,
VMill-Mill-0,   VMill-MillFeeder-125,
VMill-Idle-125,   VMill-LatheBin-125,
VMill-VLathe-250,   VMill-VMillFeeder-125,
VMill-MillBin-125,
VMill-Mill_In_Unload-25,
VMill-Mill_In_load-25,
VMill-Mill_Out_Unload-0,
VMill-Mill_Out_Load-0,
VMill-Lathe_In_Unload-250,
VMill-Lathe_In_load-250,
VMill-Lathe_Out_Unload-250,
VMill-Lathe_Out_Load-250,
VMill-Lathe-250,

Mill-VMill-0,   Mill-MillFeeder-125,
Mill-Idle-125,   Mill-LatheBin-125,
Mill-VLathe-250,   Mill-VMillFeeder-125,
Mill-MillBin-125,
Mill-Mill_In_Unload-25,
Mill-Mill_In_load-25,
Mill-Mill_Out_Unload-0,
Mill-Mill_Out_Load-0,
Mill-Lathe_In_Unload-250,
Mill-Lathe_In_load-250,
Mill-Lathe_Out_Unload-250,
Mill-Lathe_Out_Load-250,
Mill-Lathe-250,

Lathe_Out_Unload-Lathe-0,
Lathe_Out_Unload-VLathe-0,
Lathe_Out_Unload-Lathe_In_Unload-25,
Lathe_Out_Unload-Lathe_In_Load-0,
Lathe_Out_Unload-Mill-250,
Lathe_Out_Unload-VMill-250,
Lathe_Out_Unload-Mill_Out_Unload-250,
Mill_Out_Load-LatheBin-125,
Mill_Out_Load-LatheFeeder-125,
Mill_Out_Load-MillFeeder-125,
Mill_Out_Load-VLatheFeeder-125,
Mill_Out_Load-VmillFeeder-125,

LatheFeeder-LatheBin-75,
LatheFeeder-MillBin-75,
MillFeeder-MillBin-75,
MillFeeder-LatheBin-75,
VLatheFeeder-LatheBin-75,
VLatheFeeder-MillBin-75,
VMillFeeder-MillBin-75,
VMillFeeder-LatheBin-75,

Idle-MillFeeder-125,
Idle-LatheFeeder-125,
Idle-VMillFeeder-125,
Idle-VLatheFeeder-125;

LAYOUTS: "cimlab.lay",parttype,0.1,0.1,"s";
END;
C.2.1 inet.h

/********************************************************
 * This is inet.h
 * these are the general declarations for establishing
 * the connection with the server
 *********************************************************/

#define HOST_ADDR "alpha"
#define GENERIC_PORT 5004

#define MAXMSGLEN 80
#define HEADERLEN 20
#define TOTALMSGLEN MAXMSGLEN + HEADERLEN
#define HEA DFIELDLEN 5 /* %4d + space */
#define IDLEN 5
#define PINGGAP 60
#define ACCEPTTIME 3 /* 3 seconds */

/* command line standards */
#define LISTENTIMEOUT 1
#define MINCLIENTS 0
#define MAXCLIENTS 10
#define MAXTIMEOUT 1
#define MAXGENS 5
#define MAXIMPORTANTS 5

/* destinations */
#define SERVER 0
#define ALL -1

/* type */
#define NORMAL 0
#define GO 1
#define KILL 2
#define HALT 3
#define RETURNED 4
#define PING 5
#define CLOSE 6
#define NUMPARAMS 3

#ifdef DEBUG
#define PERR_EX (text, n) (\n    perror(text);\n    EXIT(n);\n  )
#else
#define PERR_EX (text, n) EXIT(n);
#endif
#ifdef DEBUG
#define PERR_RET(text,n) {
    perror(text);
    return(n);
}
#endif

#define PERR_RET(text,n) return(n);
#endif

#ifdef DEBUG
#define PERR_CONT(text) {
    perror(text);
    continue;
}
#endif

#define PERR_CONT(text) continue;
#endif
C.2.2 Register.h

/*************************************************************************/
* this connects to server using a stream socket
* in that it gets the port of stream socket of server
* now with that port number it connects to server
**************************************************************************/

int Sock;
/*************************************************************************/
* establish
* establishes the connection
* if successful returns one else returns zero
**************************************************************************/

int establish(hostname, noofids, idbuf, gen)
char *hostname;
int noofids;
char *idbuf; /* %4d %4d .... 1st for number of ids followed by ids */
int gen;
{
    struct sockaddr_in server;
    char bufptr[TOTALMSGLEN];
    struct hostent *hp, *gethostbyname();

    int sockd, re, port, i;
    int length, dest, source, type;
    extern int Sock;

    struct sockaddr_in name;

    if (hostname == NULL )
        {
            fprintf(stderr,"establish :GIVE hostname \n");
            return(0);
        }
    /* create socket on which to send */
    sockd =socket(AF_INET, SOCK_STREAM, 0);
    if (sockd < 0)
        {
            perror("Opening generic socket");
            return(0);
        }
    /* construct the name of the socket to send to */
    hp = gethostbyname(hostname);
    if (hp ==0)
        {
            fprintf(stderr, "establish : %s <- unknown host\n", hostname);
            return(0);
        }

```c
```
bcopy((char *)hp->h_addr, (char *)&name.sin_addr, hp->h_length);
nname.sin_family = AF_INET;
nname.sin_port = htons(gen);
printf("gen \n", gen);

if (connect(sockd, (struct sockaddr *)&name, sizeof (name)) < 0)
{
    perror("Connecting generic socket ");
    return(0);
}

if (read(sockd, buftpri, 80)< 0)
{
    perror("Reading generic socket ");
    return(0);
}

close(sockd);
sscanf(buftpri, "%4d", &port);

/* create socket on which to send */
Sock =socket(AF_INET, SOCK_STREAM, 0);
if (Sock < 0)
{
    perror("Opening stream socket");
    return(0);
}

/* construct the name of the socket to send to */
hp = gethostbyname(hostname);
bcopy((char *)hp->h_addr, (char *)&server.sin_addr,
    hp->h_length);
server.sin_family = AF_INET;
server.sin_port = htons(port);
printf("establish : connecting to \n", htons(port));

/* connect to server */
if (connect(Sock, (struct sockaddr *)&server, sizeof (server)) < 0)
{
    perror("Connecting stream socket ");
    return(0);
}

if (write(sockd, idbuf, strlen(idbuf)) < 0)
{
    perror("Sending request message ");
}

/* send message */
printf("sending request: (%s)\n", idbuf);
printf("Waiting for GO msg\n");
type = ALL;
i = 1;
while(i<=noofids){
    type = ALL;
    readmesg(&length,&type,bufptr);  //*** read go */
    if (type == GO) {
        printf("%dth GO\n",i);
        i++;
    }
    if (type == CLOSE)
        break;
}
printf("Establish: ending\n");
if (type == CLOSE)
    return(0);
else
    return (1);
}

/****************************Nguồn****************************
* write to the socket
*******************************************/
int writemesg(dest, source, type, bufptr )
int dest;
int source;
int type;
char *bufptr;
{
    /* get message */
    char message[TOTALMSGLEN];
    int length;
    int w;
    extern int Sock;

#ifndef PC
    if (check(Sock, CONEC)) {
#endif
    /* assumptions ---
     * string is null terminated
     * length is total message length
    */
    length = strlen(bufptr)+HEADERLEN;
    sprintf(message,"%4d %4d %4d %d %s",length,
    dest,source,type,bufptr);
    if ((w = write(Sock,message,length)) < 0)
    {
        perror(" sending message ");
        return (-1);
    }

#ifndef PC
    }
#endif


```c
#else
return(-1);
#endif

int readmesg (length, type, bufptr)
int *length;
int *type;
char *bufptr;
{
    int re = 1;
    char head[HEADERLEN+1];
    extern int Sock;
    extern int Value[CHECK];

    int re = 1;
    char head[HEADERLEN+1];
    extern int Sock;
    extern int Value[CHECK];

    #ifdef PC
    if (check(Sock, CONNEC))
        if (check(Sock, ANYDATA))
    #else
    re = check(Sock, COUNT);
    if (re>=HEADERLEN)
    #endif
    if ((re = read(Sock, head, HEADERLEN)) < 0 ){
        perror ("reading from stream header");
        return(-1);
    }
    sscanf(head, "%d", length);
    printf("readmesg: length is %4d\n", *length);
    sscanf (head, "%d %d %d %d", length,
            &Value[DEST], &Value[SOURCE], type);
    if (((re = read(Sock, bufptr,
                      *length-HEADERLEN)) < 0 ){
        perror ("reading from stream message");
        return(-1);
    }
    if ((*type == CLOSE) || (*type == KILL)) { 
        close(Sock);
        exit(0);
    }
    mycpy(bufptr, bufptr, *length-HEADERLEN,
            TOTALMSGLEN);
    }
```
return(re);
#endif

int check(fd, what)
int fd, what;
{
    int dumint = 0;
    fd_set ready;     /* for select */
    fd_set ready2;    /* for select */
    struct timeval to;

    switch (what) {
    case COUNT :
        if (ioctl(fd,FIONREAD,&dumint) < 0)
            PERR_RET("ioctl in Check COUNT",-1);
        return((int)dumint);
        break;
    case ANYDATA :
if (ioctl(fd,FIONREAD,&dumint) < 0)
    PERR_RET("ioctl in Check DATA",0);
    return(dumint>0);
break;
case CONNEC:
    FD_ZERO(&ready);
    FD_SET(fd,(struct fd_set *) &ready);
    FD_ZERO(&ready2);
    FD_SET(fd,(struct fd_set *) &ready2);
    to.tv_sec = 0;
    to.tv_usec = 1;
    if (select(FD_SETSIZE, (fd_set *)&ready,
                (fd_set *)&ready2, (fd_set *)0,
                (struct timeval *)&to) < 0)
        PERR_RET("Select in Check CONNEC",0);
    if ((FD_ISSET(fd,(struct fd_set *) &ready)) ||
        (!FD_ISSET(fd,(struct fd_set *)&ready2))){
        if (ioctl(fd,FIONREAD,&dumint) < 0)  
            PERR RET("ioctl in Check CONNEC",0);
        if (dumint == 0) {
            #ifdef DEBUG
                printf("Bad connection\n");
            #endif
            return(0);
        }
    } else
        return(1);
}
C.2.3 cimlab.h

/******************************************************************************
 * This is cimlab.h file
 * This file consists of all declarations, structures, *
 * global variables
*******************************************************************************/

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <floatingpoint.h>
#include <sys/time.h>
#include <sys/ioctl.h>

#define REQ_CLEAR 1
#define REQ_SENT 2
#define REQ_ARRIVED 3
#define CHECK 4
#define CMD 1
#define REQ 2
#define REPLY 3
#define SOURCE 0
#define DEST 1
#define MSGID 2
#define NUMPARAMS 3
#define INDEX 4
#define TRUE 1
#define MAX_OFFSET 20
#define MAX_REQ_MSG 30
#define MAX_INPUT_MSG 30
#define ANYDATA 1
#define CONNEC 2
#define COUNT 3
#define POLLTIMEOUT 15

typedef struct msg{
    int state;
    int tn;
    int entity;
    int sig;
    int source;
    char label[10];
}MSG;

struct actor{
MSG inp_msg[MAX_INPUT_MSG];
MSG req_msg[MAX_REQ_MSG];
}

struct actor *Actor;
smint lenta;
int Max_actor, Min_actor;
int Var[INDEX];
int Value[INDEX];
int Sock;
long Init_sim_time;
long Init_real_time;
char idbuf[TOTALMSGLEN];
C.2.4 cimlab.lib

/***********************************************************************
*    This file is cimlab.lib. It contains user routines
***********************************************************************/

/***********************************************************************
* Routine:        scan_param
* Description:    Called from SIMAN cevent routine to read the contents
*                 of the string "param" and assigns the values to attributes of that entity.
*                 
*                 Siman functions: seta (smint *, smint *, float *)
*                 
* Inputs:         char string buf and the entity
*                 
* Returns:        none
***********************************************************************

scan_buf(entity,buf)
char buf[HEADERLEN];
int entity;
{
    int newbuf[HEADERLEN];
    int j,n;
    float x;

    j=1;
    *newbuf = '/0';
    n=sscanf(buf,"%e[\1-\177]",&x,newbuf);
    while (n>0) {
        seta(&entity,&j,&x);
        printf("SIMAN: The %d number is =%e\n",j++,x);
        strncpy(buf,newbuf);
        *newbuf= '\0';
        n=sscanf(buf,"%e[\1-\177]",&x,newbuf);
        j++;
    }
    return;
}

/***********************************************************************
* Routine:        put_param
* Description:    Called from SIMAN cevent routine to
***********************************************************************

/put_param
* store the values of the attributes for a entity in the string "param" before sending it to the flexis or server.

* Siman functions: float a(smint *, smint *)

* Inputs: No of Attributes entity and "param"

* Returns: param

*********************************************************************************/

int put_param(entity, no_of_attrs, param)
int no_of_attrs;
smint *entity;
char param[];
{
    int val_attrib;
    int i;
    char newbuf[TOTALMSGLEN];

    *newbuf='\0';

    printf("the value of entity is %d\n",*entity);
    if(no_of_attrs==0) {
        *newbuf='\0';
        *param='\0';
        *newbuf='\0';
        return;
    }
    for (i=no_of_attrs;i=0;i--)
    {
        val_attrib=(int) a(entity,&i);
        sprintf(param,"%4d %s",val_attrib,newbuf);
        strcpy (newbuf,param);
        return;
    }
}

*********************************************************************************/

* Routine: variable_val

* Description: Called from SIMAN cevent routine to assign the values of the variables for a entity in the string "param" to Source, Dest, Msgid, Numparams before sending it to the Server.
Siman functions:
float v(smint *)

Inputs:
No of Attributes entity and "param"

Returns:
none

-------------------------------

int variable_val(index,Value,Var)
int Value[];
int Var[];
int index;
{
    int i,x;
    for(i=0;i<index;i++) {
        x=Var[i];
        Value[i]=(int) v(&x);
    }
    printf("The value of numparams is %d\n",Value[3]);
    return;
}

/*-------------------------------*/

* Routine:     getnoofids
* *
* Description: Called from SIMAN prime routine to
*     the no of actor ids and also
*     arranging the idbuf in a message
*     format %4d %4d %4d .......
* *
* Siman functions: None
* *
* Inputs:      string containg the actor ids
* *
* Returns:     noofids
* *
*-------------------------------*/

int getnoofids(idbuf)
char idbuf[TOTALMSGLEN];
{
    char newbuf[TOTALMSGLEN];
    char buf1[TOTALMSGLEN],buf2[TOTALMSGLEN];
    int i,n;
    int x;
    printf("value of new idbuf is %s\n",idbuf);
i=0;
*newbuf = '\0';
*buf1 = '\0';
n=sscanf(idbuf,"%d[1-177],&x,newbuf);
sprintf(buf1,"%4d",x);
strcpy(buf2,buf1);
while (n>0) {
  strcpy(idbuf,newbuf);
  strcpy(buf1,buf2);
  *newbuf='\0';
  *buf2='\0';
i++;
n=sscanf(idbuf,"%d[1-177],&x,newbuf);
  sprintf(buf2,"%s %4d",buf1,x);
}
strcpy(idbuf,buf1);
return(i-1);
**C.2.5 cimlab.c**

```c
/*---------------------------------------------
 * Module:      CIMLAB
 * Description: Contains user written C functions
 * Routines:    cprime - initial conditions event setup
 *              cwrap - end of replication logic
 *              cevent - event number mapped to appropriate event routine
 *              cuf - computed in user function nuf
 *              cur - index computed in user rule NUR
 *              cstate - state and differential equations
 *              cuclea - clears user-defined statistics
 *              cusav - save current status of system
 *              curst - recall saved status of system
 *              ckeyhi - key hit during simulation
 *
 * SIMLIB.H contains some definitions and the declarations necessary to call SIMAN FORTRAN routines.

#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <time.h>
#include "/usr/local/cinema1.31/lib/simlib.h"
#include ".../../include/inet.h"
#include ".../../include/cimlab.h"
#include ".../../include/cimlab.lib"
#include ".../../include/register.lib"
#include ".../../include/msg.lib"

/*---------------------------------------------
 * Routine :      cprime
 * Description :  Called from SIMAN at the beginning of each simulation replication to establish the initial conditions and schedule the initial event for the model.
 * Inputs :       simstr *sim - common block pointer
 * Returns :      none
```
#ifdef SUN_
void cprime_( sim )
#else
void cprime ( sim )
#endif
simstr *sim;
{

smint lent;
smint Event_no,number;
int index1,index2,itype,noofids;
int len,i,j;
int sourceid,msgida;
int speed;
float velocity;
float dtime;
extern char idbuf[TOTALMSGLEN];
FILE *fp;

/* opens the config file and reads the file */

printf("entered Prime \n");
fp=fopen("./cimlab.cnf","r");
number=0;
fscanf(fp,"%4d %4d\n",&Min_actor,&Max_actor);
printf("Max actor is %4d Min actor is %4d\n",Min_actor,Max_actor);
printf("entered Prime \n");
Actor=(struct actor*)malloc((Max_actor-Min_actor+2)*sizeof(struct actor));
fscanf(fp,"%[^\n]",idbuf);
fscanf(fp,"%f\n",&velocity);

while(!feof(fp)) {
  fscanf(fp,"%d %d",&sourceid,&msgida);
  fscanf(fp,"%s\n", Actor[Max_actor-sourceid+1].inp_msg[msgida].label);
  number++;
}
fclose(fp);

for(i=1;i<=Max_actor-Min_actor+1;i++) {
  for(j=0;j<=MAX_REQ_MSG;j++) {
    Actor[i].inp_msg[j].state=REQ_CLEAR;
  }
}
noofids=getnoofids(idbuf);
if(!establish("alpha",noofids,idbuf,5004)) {
    exit(0);
}

index1=0;
index2=0;
itype=29;
len=strlen("Numparams");
Var[NUMPARAMS]=nsym("Numparams", &len, &itype, &index1, &index2);

len=strlen("Msgid");
Var[MSGID]=nsym("Msgid", &len, &itype, &index1, &index2);
len=strlen("Source");
Var[SOURCE]=nsym("Source", &len, &itype, &index1, &index2);

len=strlen("Dest");
Var[DEST]=nsym("Dest", &len, &itype, &index1, &index2);
len=strlen("VELOCITY");
speed=nsym("VELOCITY", &len, &itype, &index1, &index2);
setv(&speed, &velocity);

create(&lent);
dtime=0.0;
Event_no =CHECK;
Init_sim_time=(long)sim->tnow;
Init_real_time=(long)time((time_t *)NULL);
sched(&lent, &Event_no, &dtime);
return;

******************************************************************************
* Routine : cwrap
* * Description : Called from SIMAN at the end of each simulation replication to process the logic associated with the end of the replication.
* * Inputs : simstr *sim - common block pointer
* * Returns : none
******************************************************************************

#ifdef SUN_
void cwrap_( sim )
#else
do cerr
#endif
simstr *sim;
{
extern int Sock;
int dest, source;
char bufptr[TOTALMSGLEN];

writemsg(dest, source, KILL, bufptr);
close(Sock);
exit(0);
return;
}

/****Routine: cevent****
* Description: Maps the event number n to a call to the event subroutine containing the logic for the event.
* Inputs: smint *l - index of entity being processed by the event
*         smint *n - event number
*         simstr *sim - common block pointer
* Returns: none
*********

#ifdef SUN_
void cevent_( l, n, sim )
#else
void cevent( l, n, sim )
#endif

smint *l;
smint *n;
simstr *sim;
{
extern int Value[INDEX];
float dtime;
int source, type, tn, length;
int re;
int event_no;
char bufptr[TOTALMSGLEN];

switch (*n) {
    case CMD :
send_cmd(1);
  break;

case REQ :
  send_req(1);
  break;

case REPLY :
  send_rep(1);
  break;

case CHECK :
  re = readmesg(&length,&type,bufptr);
  if (re> 1) {
    printf("%s\n",bufptr);
    process_msg(Value,bufptr);
  }
  event_no=CHECK;
  dtime=1.0;
  sched(1,&event_no,&dtime);
  synchronize_time(sim);
  break;
}
return;

int process_msg(Value,bufptr)
int Value[];
char bufptr[TOTALMSGLEN];
{
  int type;
  sscanf(bufptr,"%4d",&type);

  switch(type) {
    case CMD :
      receive_cmd(bufptr);
      break;

    case REQ :
      receive_req(bufptr);
      break;

    case REPLY :
      receive_rep(bufptr);
      break;
  }
  return;
}
C.2.6 msg.lib

/ *************************************************************/
* This is msg.lib file. The message routines are described  *
* here
/ *************************************************************/

Routine : send_cmd
*
* Description: Called from SIMAN cevent routine to send    *
*        the COMMAND message.
*
* Siman functions: none
*
* Inputs : entity integer.
*
* Returns : none
/ *************************************************************/

send_cmd(1)
smint *l;
{
   extern int Value[INDEX], Var[INDEX];
   char buf[TOTALMSGLEN];
   char param[TOTALMSGLEN];

   variable_val(INDEX, Value, Var);
   put_param(&l, Value[NUMPARAMS], param);
   printf("%s\n", param);
   sprintf(buf,"%4d %4d %s", CMD, Value[MSGID], param);
   writemsg(Value[DEST], Value[SOURCE], NORMAL, buf) ;
}

/ *************************************************************/
* Routine : send_req                                     *
*
* Description: Called from SIMAN cevent routine to send    *
*        the REQUEST message.
*
* Siman functions: none
*
* Inputs : entity integer.
*
* Returns : none
/ *************************************************************/

send_req(1)
smint *l;
{
   extern int Value[INDEX];
extern int Var[INDEX];
extern int Max_actor;
int tn, found, len, itype;
int index1, index2, sig_no, sig;
char param[TOTALMSGLEN];
char buf[TOTALMSGLEN];

variable_val(INDEX, Value, Var);
Value[SOURCE] = Max_actor - Value[SOURCE] + 1;
for (tn = 0; tn <= MAX_REQ_MSG; tn++) {
    if (Actor[Value[SOURCE]].req_msg[tn].state ==
        REQ_CLEAR)(
        found = TRUE;
        break;
    }  
}  
if (found != TRUE) {
    printf("Siman: %4d -- Cannot find a MESG_CLEAR\n", Value[SOURCE]);
    exit(0);
}  
Actor[Value[SOURCE]].req_msg[tn].state = REQ_SENT;
len = strlen("Sig");
itype = 28;
index1 = 0;
index2 = 0;
sig_no = nsym("Sig", &len, &itype, &index1, &index2);
sig = a(l, &sig_no);
Actor[Value[SOURCE]].req_msg[tn].sig = sig;
Actor[Value[SOURCE]].req_msg[tn].entity = *l;
p1t_param(&l, Value[NUMPARAMS], param);
printf("%s\n", param);
Value[SOURCE] = Max_actor - Value[SOURCE] + 1;
sprintf(buf, "%4d %4d %4d %s", REQ, tn, Value[MSGID],
        param);
write mesg(Value[DEST], Value[SOURCE], NORMAL, buf) ;

/*****************************************
* Routine: send_rep
* Description: Called from SIMAN cvent routine to send
* the REPLY message.
* Siman functions: none
* Inputs: entity integer.
* Returns: none
*******************************************/
send_rep(1)
smint *l;
{
    extern int Value[INDEX];
    extern int Var[INDEX];
    int tn, destination;
    char param[TOTALMSGLEN];
    char buf[TOTALMSGLEN];

    variable_val(INDEX, Value, Var);
    Value[SOURCE] = Max_actor-Value[SOURCE] + 1;
    Actor[Value[SOURCE]].inp_msg[Value[MSGID]].
    state = REQ_CLEAR;
    tn = Actor[Value[SOURCE]].inp_msg[Value[MSGID]].tn;
    put_param(l, Value[NUMPARAMS], param);
    destination = Actor[Value[SOURCE]].inp_msg
        [Value[MSGID]].source;
    Value[SOURCE] = Max_actor-Value[SOURCE] + 1;
    sprintf(buf, "%4d %4d %4d %s", REPLY, tn, Value[MSGID],
            param);
    writemsg(destination, Value[SOURCE], NORMAL, buf);
}

/*****************************************************************
* Routine: receive_cmd
* Description: Called from SIMAN event routine to
* process the COMMAND message.
* Siman functions: send()
* Inputs: char string bufptr.
* Returns: none
******************************************************************/

receive_cmd(bufptr)
char bufptr[TOTALMSGLEN];
{
    extern int lenta;
    extern int Value[INDEX];
    int slen, type;
    float dtime=0.0;
    char param[TOTALMSGLEN];
    sscanf(bufptr, "%4d %4d %4d %s", &type, &Value[MSGID], param);
    create(&lenta);
    scan_buf(lenta, param);
    Value[DEST] = Max_actor-Value[DEST] + 1;
    slen=strlen(Actor[Value[DEST]].
                inp_msg[Value[MSGID]].label);
    send(&lenta, &dtime, Actor[Value[DEST]].

Routine: receive-req

Description: Called from SIMAN event routine to process the REQUEST message.

Siman functions: send()

Inputs: char string bufptr.

Returns: none

receive-req(bufptr)
char bufptr[TOTALMSGLEN];
{
    extern int lenta;
    extern int Value[INDEX];
    extern int Max_actor;
    int slen,type,tn;
    float dtime=0.0;
    char param[TOTALMSGLEN];

    sscanf(bufptr,"%4d %4d %4d %s",&type,&tn,
           &Value[MSGID],param);
    Value[DEST]=Max_actor-Value[DEST]+1;
    Actor[Value[DEST]].inp_msg
        [Value[MSGID]].state=REQ_ARRIVED;
    Actor[Value[DEST]].inp_msg[Value[MSGID]].tn=tn;
    Actor[Value[DEST]].inp_msg[Value[MSGID]].source
        =Value[SOURCE];
    create(&lenta);
    scan_buf(lenta,param);
    slen=strlen(Actor[Value[DEST]].inp_msg
                [Value[MSGID]].label);
    send(&lenta,&dtime,Actor[Value[DEST]].
         inp_msg[Value[MSGID]].label,&slen);
}

Routine: receive-rep

Description: Called from SIMAN event routine to process the REPLY message.

Siman functions: send()

Inputs: char string bufptr.
receive_rep(bufptr)
char bufptr[TOTALMSGLEN];
{
    extern int lenta;
    extern int VALUE[INDEX];
    int ent_release =1;
    int go_signal,tn,type;
    char param[TOTALMSGLEN];

    sscanf(bufptr,"%4d %4d %4d %s", &type,&tn,
          &Value[INDEX],param);
    lenta=Actor[Value[INDEX]].req_msg[tn].entity;
    scan_buf(lenta,param);
    Actor[Value[INDEX]].req_msg[tn].state=REQ_CLEAR;
    go_signal=Actor[Value[INDEX]].req_msg[tn].sig;
    signal(&go_signal,&ent_release);
}

/******************************/
* Routine : synchronize_time
* Description: Called from SIMAN event routine to synchronize the simulation time with the real time.
* Siman functions: Tnow.
* Inputs : simstr *sim.
* Returns : none
******************************/

int synchronize_time(sim)
simstr *sim;
{
    unsigned diff_time;
    long diff_sim_time;
    int time_delay_on =1;

    diff_sim_time=(long)sim->tnow - Init_sim_time;
    if(time_delay_on)
        while(((long)time(NULL)-Init_real_time)<=diff_sim_time)
            diff_time=((long)time(NULL)-Init_real_time)-(diff_sim_time);
        usleep(diff_time);
if(((long)time((time_t *)NULL)-Init_real_time)>=diff_sim_time+MAX_OFFSET){
    printf("\nERROR\n");
    exit(0);
}
C.2.7 Makefile

PEXT=F
FFK=f77
LIB1=cs1S4.a
LIB2=cs2S4.a
MACH=SUN_
CC=cc -c
modules=millmodule+lathemodule+robotmodule
CP=cp
RM=rm
exename=cimlab
MODEL=/usr/local/siman/model
EXPMT=/usr/local/siman/expmt
LINKER=/usr/local/siman/linker
CIMLAB.LIB=../include/cimlab.lib
MSG.LIB=../include/msg.lib
CIMLAB.H=../include/cimlab.h
REGISTER=../include/register.lib
INET=../include/inet.h

all: millmodule.m lathemodule.m robotmodule.m cimlab.e
cimlab.p remove cimlab
millmodule.m: millmodule.mod
   $(MODEL) millmodule.mod millmodule.m
lathemodule.m: lathemodule.mod
   $(MODEL) lathemodule.mod lathemodule.m
cimlab.e: cimlab.exp
   $(EXPMT) cimlab.exp cimlab.e
robotmodule.m: robotmodule.mod
   $(MODEL) robotmodule.mod robotmodule.m

millmodule.mod cimlab.e
   $(LINKER) $(modules) cimlab.e cimlab.p
   $(CP) cimlab.p ../bin
remove: $(modules)
   $(RM) $(modules)

millmodule+lathemodule+robotmodule: millmodule.m
   $(CP) millmodule.m millmodule+lathemodule+robotmodule
cimlab: cimlab.c $(CIMLAB.LIB) $(CIMLAB.H) $(MSG.LIB)
   $(REGISTER) $(INET)
   $(CC) -D$(MACH) cimlab.c
   $(FFK) -o $(exename) -Bstatic $(exename).o
/usr/local/cinema1.31/lib/userforc.o
/usr/local/cinema1.31/lib/cmain.o
/usr/local/cinema1.31/lib/csiman.o
/usr/local/cinema1.31/lib/$LIB1
/usr/local/cinema1.31/lib/$LIB2 -lx11
   $(CP) $(exename) ../bin